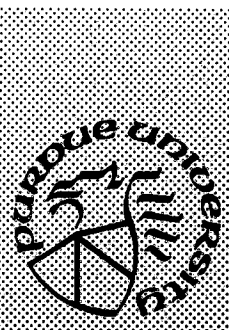
12-1-1989

Compiler-Driven Cache Management Using A State Level Transition Model

Chi-Hung Chi
Purdue University



Compiler-Driven Cache Management Using A State Level Transition Model

Chi-Hung Chi

TR-EE 89-71
December 1989

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

COMPILER-DRIVEN CACHE MANAGEMENT USING A STATE LEVEL TRANSITION MODEL

A Thesis

Submitted to the Faculty

of

Purdue University

by

Chi-Hung Chi

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 1989

this is dedicated
to my dearest Lai-Fan

ACKNOWLEDGMENTS

I wish to express my most sincere appreciation to my major professor, Dr. Hank Dietz, for his patience and guidance throughout the course of this research. He taught me how to do research, helped me expand my research sights, and spent time discussing crazy ideas with me. He is also one of my best friends, who always gives me support, love, and encouragement whenever I need them.

I would like to extend my thanks to Dr. Dave Meyer and all members of the Compiler-oriented Architecture Research Group at Purdue (CARP) for their suggestions. The support from my parents, Mr. and Mrs. Sau Nam Che is one of the key elements to allow this thesis to be completed.

Finally, I dedicate this thesis to my dearest wife, Lai-Fan, for all her unconditional love and support during the course of my graduate study. Initially, I thought I could spend more time with her once this thesis was finished. However, she could not wait for this time to come and she went to be with the Lord Jesus Christ. I know she is very happy because she is with Jesus Christ, which is far the best. However, I miss her and look forward to see her again in heaven.

TABLE OF CONTENTS

Page

Page

Page

Page

Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

ABSTRACT

Chi, Chi-Hung Purdue University. May 1989. Compiler-Driven Cache Management Using a State Transition Model. Major Professor: Hank Dietz.

Cache performance is critical in cache-based supercomputers, where the cache-miss/cache-hit memory reference delay ratio is typically large. Using compile-time analysis, program behavior can be predicted, and cache control directives can be embedded in the generated code. Thus, cache performance can be improved in a way not possible using conventional techniques. Given hardware able to selectively bypass the cache, cache performance can be increased because pollution can be minimized. Cache line replacement can also be controlled by the compiler (rather than by LRU, etc.), further enhancing performance. The research consists of the development of a model and algorithms providing optimal, or near optimal, cache performance by compiler management of cache.

CHAPTER I
MOTIVATIONS AND SCOPE

1.1 Introduction

With advances in VLSI and supercomputing, a complex high-performance processor with computational capability similar to a supercomputer can be fabricated on a single Very Large Scale Integration (VLSI) chip. Although improvement in integrated circuit technology has resulted in significantly reduced gate delays, the speed and density of memory parts have not improved proportionately. Consequently, the overall performance of computers using these processors is usually limited by the memory system speed. Cache is introduced to bridge the gap between the memory and processor speeds.

Cache memory is a high speed buffer memory between the Central Processing Unit (CPU) or multiprocessor Processing Element (PE) and the main memory. The basic idea is to maintain quickly-accessible copies of the data and instructions which are most likely to be needed by the processor. Since the cache memory can be much smaller than main memory, it is feasible to implement cache memory using faster, more expensive, technology than is used to implement main memory; if the cache holds the right data and instructions, the processor effectively sees the fast cache access time, yet has the large main memory address space. Since the first implementation of cache in the IBM 360/85 computer in 1969 [Lip68] [Smi87b], the ability of cache memory to cost effectively improve system performance has been widely accepted. Significant reductions in the average data/instruction access time have been achieved using very simple cache placement/replacement policies implemented in hardware [Bel74]. For this reason, most current computer systems us some type of cache.

Due to the limited area for an cache on-chip in VLSI, the high ratio of on-chip/off-chip reference delay, and the increasing demand for faster and larger memory, simple hardware cache placement/replacement policies which were developed in the early 1970's are no longer sufficient to bridge the

ever-increasing "memory reference delay gap" between the processor and the main memory [HiS84] [PaS82] [MiF86].

In the past few years, the rapid growth of supercomputing and VLSI has provoked the re-evaluation of many computer design concepts (e.g. compiler methods for optimization and parallelization [Die87], architecture concepts of RISC and CISC [Pat85]), hence inspiring many new and important innovations. However, the traditional memory hierarchy — especially cache design — seems to have escaped this re-evaluation; there is a general consensus that caches cannot be greatly improved, perhaps because such attempts were essentially fruitless (improvements of only a few percent) over approximately the last 5 years.

It is true that numerous research efforts have been devoted to cache design in multiprocessor environments. The cache consistency problem is the most common focus. Yet, very little effort is spent in re-evaluating basic cache design concepts and the assumptions behind them to determine if these constraints and assumptions are still valid and useful.

A good example is Belady's MIN algorithm [Bel66], which was proposed in 1966. At that time, because of the immaturity of compiler technology, the MIN algorithm was classified as the "unrealistic optimal" policy (hence, the name MIN). As understanding of compiler optimization and architecture improved, it became feasible to implement MIN algorithm. However, MIN is not optimal in the sense that it does not minimize execution *time*, but rather minimizes a quantity which was thought to correspond to execution time. Even today, this flaw in MIN is not widely noted.

In order to have a major breakthrough in cache design, an "intelligent" cache management scheme — based on the current understanding of computer architecture and compiler analysis techniques — is necessary.

1.2 Motivation

As cache performance becomes more critical in determining the overall system performance, more research is devoted to cache design. However, most of this effort is spent in "cache design for a limited domain within a particular machine" rather than in "basic cache research" — research on the basic principles of cache and to improve cache performance by fine-tuning these principles.

Major focuses of current cache research efforts include:

- Cache size

  Since cache is "never large enough," this research focuses on designing a larger on-chip cache. Another alternative approach for this is to build external cache chips [Hil83] [HiS84].

- Additional features

  In order to enhance cache performance, this research tries to append extra features to traditional cache, hoping that these new features will increase system performance. For example, since one limitation of cache chip is the I/O bandwidth, it is seems wise to add an extra I/O port in the chip design to increase off-chip bandwidth [Smi82].

- Cache prefetch

  Cache prefetch is generally viewed as a good way to improve cache performance, provided that the prefetched information does not pollute cache. Hence, various possible cache prefetch mechanisms based on guessing and/or program flow analysis are studied [Smi78a] [Smi78b] [Smi82].

- Simulation

  Most caches are currently designed by simulating many designs and choosing the "best": a very time-consuming procedure. Large simulations are performed to select a set of "good" cache design parameters (e.g. cache hardware parameters, cache management policies) for a limited application domain, using a few benchmark programs [Smi82].

- Cache consistency

  In a multiprocessor environment, an additional problem for cache design is to make sure that copies of the same data stored in different caches (one for each processing element) are consistent with each other. This research focuses on reducing the cost of maintaining consistency among processing elements [Goo83] [LeY87].

However, these research topics merely extend current cache design guidelines — helping to select a good set of parameters and policies from existing cache design concepts for a particular application area. The validity of constraints and assumptions behind these cache design concepts are neither modified nor questioned.

Some fundamental cache design issues that are often forgotten or overlooked in cache research, yet which can greatly improve performance if they are handled properly, are:

- Optimality of cache performance — what is the best cache performance conceptually possible?
- Cache pollution — what information should *not* be in cache?
- Cost consideration — can a cache policy be based directly on minimizing execution time, as opposed to minimizing some measure which sometimes correlates with execution time?
- Cache hardware structure — what really needs to be controlled and by what mechanism?
- Generality of memory hierarchy — how do registers, cache, virtual memory, etc. all fit together?
- VLSI on-chip cache — how can one make best use of very small, very fast, caches?

In the next six subsections, each of these issues is discussed in detail. The research presented in this thesis is the result of re-evaluating and finding new answers to these basic questions of cache design and use.

1.2.1 Optimality of Cache Performance

In all previous cache management schemes, cache performance is only described in a relative sense. For example, LRU performs better than random access by 10 percent . . . but how well does the random access perform? Although Belady's MIN algorithm [Bel66] is generally considered to be the optimal cache algorithm, it is not optimal if machine state transitions and associated operation costs are taken into consideration. Since cache performance can only be compared rather than viewed against an absolute scale, it is not possible to know how much a particular cache design's performance may be able to be improved. This information is critical; since the general concensus had been that caches worked very well, few people questioned how much better they could work. The primary motivation of this thesis was the realization that cache performance was far below that which theoretically could be obtained.

There is always an argument about the best cache performance: the best performance is obtained by having all references in cache, provided that

the cache is *infinitely large.* However, there are a couple of flaws in this argument:

- It is not possible to have an infinitely large cache. What is given is only a *small* finite-size cache.
- Even if there were infinitely large caches, it would not always be desirable to make all references in cache (unless it is assumed all information is initially placed in cache without any overhead, which is not realistic). There is an overhead associated with each line being placed in cache. Unless the benefit gained by having a line in cache is greater than the overhead of cache line placement, the line should not be put in cache even if cache space is available.

As a result, a method of defining (or accurately estimating) the *true* upper bound performance of cache is needed. Knowing the optimal (or nearly optimal) cache performance can also help designers to improve cache performance based on the current methodologies.

1.2.2 Cache Pollution

A cache is said to be **polluted** if a line that has been fetched into the cache is replaced before it is referenced, or its reference frequency before it is replaced is not high enough to justify its residence in cache. Cache pollution is one of the main problems in achieving good cache performance. It becomes more serious in VLSI cache design where the area available for an on-chip cache is small and the off-chip to on-chip access time ratio is large.

Placing a new line in cache may or may not be beneficial to the cache performance because:

- there is some overhead associated with placing a line in cache,
- the residency of a line in cache might reduce the chances for other lines to use that cache space occupied by the fetched line,

However, traditional cache management schemes based on either execution history or probability are incapable of deciding when a line is referenced and what the costs and savings involved in this process are. This is particularly important in cache design with cache prefetch because bad management of prefetch simply ensures cache pollution, with no benefits. A way to control cache pollution is needed for high performance cache, with or

without cache prefetch.

1.2.3 Cost Considerations

Previous cache management schemes do not take into account the various costs of performing different cache activities. This is particularly important for multiprocessing environments where use of the memory hierarchy is often the key bottleneck. The time to fetch a datum from different memory hierarchy levels and memory locations might range from a few cycles to over a hundred cycles. Also, being able to bypass the cache is a good and cheap way to improve cache performance.

Since previous cache management schemes only consider cache-hit ratio as the major optimization issue, it does not consider the cost of each cache miss. However, if high performance from the cache is desired, this cost must be minimized.

1.2.4 Cache Hardware Structure

A lot of previous cache research focuses on the selection of parameter values such as cache size and line size. However, most of this research is based on cache hardware structure that was proposed twenty years ago. There is very little research focused on the exploration of the basic cache hardware structure.

A natural outcome of exploring new cache management schemes is the demand for new cache hardware structures that solve problems created by these new schemes. An example is the CReg structure [DiC88a] (also discussed in Section 4.7), which is a hardware solution to ambiguous data alias problem — the main problem for the MAST model described in this thesis.

1.2.5 Generality of Memory Hierarchy

In a typical computer design, the memory hierarchy usually consists of three levels, each of which is managed by different schemes:

- Registers managed by compiled-time register allocation [Fre74] [ChA81] [Cha82] [Cho83] [ChH84].
- Cache managed by hardware cache policies (e.g. replacement) [Smi82] [Smi887b].
- Main memory managed by runtime virtual memory schemes [Den70].

There are a lot of similarities among these memory hierarchy levels. Although numerous management schemes or algorithms have been developed for each of these levels, few of the results are applied across different levels. Each level has its own "independent" management scheme, and the existence of other levels is often ignored. The independent treatment of these levels introduces inefficiency in system performance.

A more general memory management scheme — that considers more than one level — is necessary. Our research shows that register allocation, cache management, and virtual memory management can employ the same model [ChD88] (of course, some minor modifications are needed).

1.2.6 VLSI On-Chip Cache

As VLSI on-chip functionality grows, there is an increasing demand for larger on-chip caches to minimize the large penalty of going off-chip for memory access [Hil83] [PaG83]. However, the on-chip area that is available for cache is always insufficient. Just modifying the set of hardware cache parameters or adding some more off-chip cache chips does not solve this problem.

On the other hand, with the maturity of compiler techniques, it is possible to improve cache performance with the help of program flow analysis [DiC88a]. For example, with the bypass cache (discussed in Section 5.4) and program flow analysis, an improvement of 10 to 30 percent can easily be obtained with very little additional hardware. The major cost is the increased compile time.

1.3 Scope of This Work

The goal of this research is to develop new cache management schemes which can solve some (if not all) of those basic cache design problems described in Section 1.2. The strategy that we use in this research is to consider *simultaneously* both the cache hardware design and the compiler analysis of programs. This strategy, called the **compiler-driven cache management**, is based on the fact that there is some significant fraction of system performance which can only be achieved by considering both hardware and software together and cannot be obtained by either of these two alone.

In this thesis, a compiler-driven cache management scheme, which is based on machine-level state transitions, is proposed as the model for control of cache activities. Algorithms are developed to obtain an optimal or nearly optimal cache allocation sequence for each program (based directly on expected total memory access time). Applications of this model to various program structures, cache organizations, and traditional cache design are also discussed. These applications are used to illustrated why optimal system performance cannot be obtained in traditional cache design and how most of this performance might be regained by modifying some fundamental cache design concepts.

Simulation results for this new model using various parameters are performed. These simulations demonstrate the validity of the above claims of large performance improvement.

Although many issues are addressed, the main focus is development of a complete[1] cache management scheme for cache control in uniprocessors.

1.4 Thesis Organization

The organization of this thesis can be outlined as follows:

- Chapter 1 is an introduction. The motivation and scope of this research are described.
- Chapter 2 gives a brief survey of previous cache research. The main emphasis of this survey is on:

[1] The word "complete" means that the methodology developed is capable of handling all common program structures, including branches and loops.

- the rationale of cache usage.
- the working principles of traditional cache design.
- the cache design guidelines that are currently used, and
- the constraints and limitations that these guidelines impose on system performance.

This survey also serves as a basis for comparison with new cache design guidelines obtained from this new model.

- Chapter 3 describes the basic model for compiler-driven cache management using machine-level state transition — the **MAST** model.
- Chapter 4 extends the MAST model to various program structures such as loops and branches. Implementation of the MAST model and various pruning techniques to reduce the computational complexity of the MAST model are also discussed. Finally, a new cache-like structure, called the CReg, is discussed in this chapter as a hardware solution to ambiguous data references.
- Chapter 5 applies the MAST model to traditional cache architectures. Operational differences between traditional cache design and MAST model are first identified. Then, modifications of traditional cache design to include some key features of the MAST model are proposed.
- Chapter 6 presents simulation results of the MAST model described in Chapter 3 and 4 and some "simplified versions" of the MAST model described in Chapter 5. These simulation results are analyzed to show the improvement of system performance obtained from the MAST model and to determine some new guidelines for cache design using the MAST model.
- Chapter 7 extends the MAST model to register allocation — the next level up in the memory hierarchy. A unified register/cache management model is proposed.
- Chapter 8 concludes this thesis by summarizing major contributions which the MAST model brings to register allocation and cache management. Future research directions are also discussed.

1.5 Definition of Terms

In order to present ideas in this thesis as clearly as possible, it is necessary to define precisely those terms which are frequently used:

Reference program:

A reference program is the skeleton of a program which contains only information relating to the memory reference pattern and control flow of a program. The reference program is used instead of the original program in the discussion of cache management and register allocation because we are only interested in the memory reference behavior of a program and not its output values. The word "reference" is sometimes omitted if its meaning in the context is clear.

Line:

A line is the amount of information (in words) that might possibly be associated with a cache tag. This is also the amount of information transferred between the main memory and the cache in a single operation.

Cache line:

A cache line refers to a line residing *in cache.*

Set:

A set is a group, or a collection — the usual meaning in Mathematics.

Cache set:

A cache set is a group of cache lines which can be associatively searched when a request for information is made to the cache. This is also the number of distinct cache line cells in which a copy of any given main memory line may reside.

Cache performance:

Cache performance refers to the total reference time used to execute a program. Note that this might not be related to the cache hit ratio, which refers to probability of finding a requested item in cache. The word "cache" is sometimes omitted if its meaning is apparent from the context.

Information:

Information refers to the items referenced in a reference program. It can refer to data and/or instructions.

Cache configuration:

Cache configureation refers to the information that is stored in cache.

CHAPTER II
EXISTING MODELS AND MANAGEMENT TECHNIQUES

2.1 Introduction

In this chapter, a brief survey of current research in cache design is presented. The main purpose of this chapter is not to give readers a brief introduction to cache design[1] rather, the intent is to explore the original design motivations, and the limitations of different cache design guidelines and cache management techniques.

This "fundamental" information about cache design is very important because it can provide hints about where and how to improve cache performance. As will be discussed in Chapter 3, 4, and 5, features which cause the poorest performance in current cache management schemes are exactly the situations where the compiler-driven cache management using MAchine level State Transitions (MAST) model is most effective in improving cache performance.

Another purpose of this chapter is to provide as a basis for comparison with cache design guidelines suggested by the MAST model. As is dicsussed later in this thesis, some cache design guidelines based on the MAST model are very different from (or even opposite to) those currently used.

In Section 2.2, the working principles which cache design is based on, as well as the basic assumptions made (either explicitly or implicitly), are discussed. Major optimization issues in current cache design are also included in this section. Section 2.3 discusses different aspects of cache design. The main focus of this section is on the guidelines for selecting cache management schemes and cache parameters. Recent research on software cache management is presented in in Section 2.4. Finally, this chapter is summarized in Section 2.5.

---

[1] An introduction to cache design can be found in most computer architecture textbooks [HwB84] [Sto87]. Detailed surveys of cache designs appear in [Smi82] [Smi87b].

2.2 Why and How Cache Works — Assumptions

There are several basic assumptions and features that are common to almost all cache designs and which designers generally accept without question. Program reference behavior is also assumed to follow some rules with respect to time and space. All these together form the current understanding of why and how a cache works. A thorough understanding of these assumptions and rules is necessary because they not only suggest how and why cache works, but also explain why cache does not always work as expected.

2.2.1 Locality of Reference

The success of cache memory has been explained by the **principle of locality of reference** in a program [Den68]. Over short periods of time, memory references made by a reference program are not uniformly distributed over the address space. Instead, references often occur in small clusters — the working set of references. There are two types of locality of reference: **temporal locality** (locality of time) and **spatial locality** (locality of space) [Den72] [Spa74] [Spi77].

The first type of locality, the temporal locality (or locality of time) refers to the fact that information currently being used is likely to be used again in the near future. This is true because most programs attempt to use code more than once during program execution[2] and the code re-used is not evenly distributed in its address space — it comes in smaller clusters. This type of reference behavior can be found in program loops in which both data and instruction references are expected to be used again for each loop iteration.

The address space usually is grouped into a fairly small number of individual, contiguous, segments of that space and only a few segments are repeatedly referenced. The second type of locality, spatial locality suggests that the loci of references of a program in the near future are likely to be near the current loci of reference. This type of reference behavior is common in scientific programs because instructions are mostly executed in a

[2] A good example is loop execution, where a loop body is repeatedly referenced. This explains why static program code size is much less than its dynamic code size (by a factor of 100 or 1000).

sequential order and it is common to access sequences of array elements (which are typically contiguous in memory).

With these localities of time and space, the rationale of cache usage is to *attempt keeping information which the processor is currently working on (local in time), together with information near the current loci of references (local in space), in cache and readily available.*

These principles of reference localities often provide good hints about what should be kept in cache. However, these locality principles are not sufficient if the cache is not large enough to hold the current working set. Moreover, when the current working set of references changes, these locality principles actually work to *decrease* cache performance.

A good example is the last use of some value i in a program. Although i is no longer needed by the program and should be first to be replaced in cache, the principle of locality of reference suggests that the value i is going to be re-used soon and hence should be kept in cache.

2.2.2 Basic Design Constraints

When cache was first proposed and implemented in the mid 1960s, there were several assumptions made about cache design based on the compiler techniques and computer architecture concepts available at that time. Since then, cache has often been designed using these assumptions without any re-evaluation or questioning of their applicability and validity. This ignores the fact that compiler techniques and architecture/hardware technologies have advanced considerably in the past twenty years.

Some fundamental assumptions made in most cache designs and the resulting problems areas follows:

- *Hardware control*:

  Assumption:

  Cache should be *purely hardware controlled* because it needs to be managed as fast as, or faster than instruction execution by the CPU and software cache control would only slow down both cache management and CPU execution [Smi82].

  Problems:

  Due to this assumption, all commonly used cache management schemes ignore the possibility of controlling cache by hardware and

software *together.* However, it is definitely feasible and desirable to use information about program reference behavior, which is collected at compile-time, to help cache hardware manage the cache at runtime. Furthermore, while it might be undesirable to execute one or more explicit cache control instructions for each memory reference, *selective use* of explicit cache control instructions should improve cache performance.

- *Knowledge of future references*:

  Assumption:

  Knowledge about future references in a reference program *is not known (or available) to cache hardware.* It is believed that cache should be managed purely in hardware. Since a hardware cache controller can only execute what is provided to it, it does not have any capability of understanding a program and determining what is going to be referenced next. This is the reason all common cache replacement policies are either based on history of execution or a random strategy (Markov Chain); it is also why Belady's MIN algorithm [Bel66] is generally referred to as the "unrealistic optimal" algorithm.

  Problem:

  A direct consequence of this assumption is that the memory reference behavior of a program is completely unknown to a hardware cache management scheme before program execution. While it is difficult (or even impossible) to determine the exact memory reference behavior of a program at compile-time, the number of possible alternatives is often small and can easily be determined by static program flow analysis (the same type of analysis used in optimizing compilers). This information about program reference behavior offers great potential to improve cache performance.

- *Cache utilization*:

  Assumption:

  Since the speed of access from cache is much faster than that from main memory, it is believed that putting information in cache *always improves system performance.* Consequently, maximizing cache utilization (i.e., to have information in cache as frequently as possible) becomes the main goal in cache design [Smi82].

Problem:

This assumption about the relationship between system performance and cache utilization is not always valid. To reference information from cache, there is always a large overhead for placing information in cache before it can be referenced. Unless the benefit obtained by placing information in cache is greater than the overhead, it might be better *not to use the cache* and make those references directly from main memory (i.e. to deliberately reduces cache utilization). In other words, cache utilization *actually decreases* to improve system performance. Moreover, since cache size is limited, the use of cache by some information might reduce the chance (or increase the overhead) for other information to use the cache.

- *Cache through*:

Assumption:

Due to the rule of "maximizing cache utilization", and the principle of locality of reference described in the last sub-section, *any reference needs to be fetched into the cache before it can be referenced by the CPU* [Smi82] (Some machines can reference a datum and transfer the corresponding block of data at the same time. However, the constraint about the residency of the block containing the referenced datum in cache still holds).

Problem:

Due to this constraint, information is always fetched into the cache upon cache miss, disregarding its effect to the system performance. As will be discussed in Chapter 6, being able to selectively bypass the cache and to reference directly from main memory can easily improve cache performance by 10 to 20 percent.

By removing these assumptions, as is done in the MAST model, a large fraction of performance lost in traditional cache management schemes can easily be regained by simple modifications to both cache hardware and cache management policies.

2.2.3 Optimization Parameters

With the reference locality principle and the basic assumptions made in cache design (as is discussed in the last subsection), there are at least five parameters that all cache designs try to optimize [Smi82], some of which conflict:

- *Hit ratio:*

  Cache hit ratio refers to the probability of finding the referenced information in cache without transferring information between the cache and the main memory first. This is a direct consequence of the *principle of maximizing cache utilization* discussed in Section 2.2.2. An indirect implication of this issue is one wants a cache as large as possible so that all information can be placed in cache without any replacement.

- *Access time for cache hit:*

  The **access time for a cache hit** refers to the average time it takes to access a datum in cache. The design rule here is to have a smaller access time for cache hit so the system performance can be improved. This has two important implications:

  - Cache control logic should be as simple as possible.
  - Cache with *size larger than necessary* might decrease the cache performance. Cache access times increase with increases in memory size because row and column access times become larger. Cache control logic involved is also more complicated for large cache. Hence, longer cache access times result.

- *Penalty for cache miss:*

  The **penalty for a cache miss** refers to the time it takes to tranfer a line of information between the main memory and cache when a cache miss occurs. This penalty depends mainly on two factors:

  - the speed of information transfer between the cache and the main memory.
  - the line size — number of elements within a line (basic information transfer unit).

- *Consistency in a multiprocessor cache:*

  Cache consistency refers to the problem of maintaining different copies of the same piece of data in different processors. It is one of the biggest problems in multiprocessor cache design because many CPU

cycles are lost in order to maintain data consistency [Vei85] [Vei86].

- *Reduction of information demanded:*
  It is desirable to reduce the total amount of information demanded in a multiprocessor system so that the bus traffic and the request queues in main memory are reduced.

These optimization issues are inter-related and it is foolish to optimize one issue without considering its impact on other issues. For example, for a fixed cache size, a change in cache line size might increase *both* the cache hit ratio and the penalty cost for cache miss. Whether the system performance is actually improved or degraded is unknown unless more information is provided.

The biggest surprise from this research is that *the cache hit ratio is not a valid optimization issue* because the cache hit ratio does not necessarily correspond to execution time. Instead, expected execution time should be used.

2.3 Cache Design Issues

There are many design issues (e.g. line size, replacement policy, etc.) that can directly affect cache performance. These issues are often inter-related — changing one might affect many others. A thorough understanding of these design issues can help selecting a good set of parameters and avoiding any negative side-effects that might impact other issues.

Basically, these issues can be classified into four groups:

- *Hardware parameters:*
  Any issue that imposes constraints to cache hardware organizations belongs to this group [Smi82] [Kab87] [MiF87] [Mit86a] [Mit86b] [PrH88]. This group includes cache hardware parameters (e.g. cache size, line size and set size) and cache partitioning (e.g. data vs instruction, user vs supervisor).
- *Management schemes:*
  Any scheme that is used to control when and how data are maintained and transferred between main memory and cache is classified in this group [Smi82] [Bab82] [Hil83] [HwB84] [Puz85] [SoR88]. This includes fetch algorithm, replacement policy, and update policy.

- *Optimizing parameters:*

  This group includes different system performance aspects that cache designers want to optimize [PoA83] [Hil83] [Mit86a] [Mit86b]. Examples are total memory access time, cache hit ratio, and bus traffic ratio.

- *Others:*

  Any other issues that are not belonged to any of the above are included here [Bre87] [ChS82] [GoH86] [LeY87]. Examples are cache consistency, and physical vs. virtual addresses.

Each of these cache design issues is discussed in detail in the next few sub-sections. Guidelines for selecting the right parameters and schemes, as suggested by current cache research, are given.

2.3.1 Hardware Cache Parameters

In this sub-section, selection of the following hardware cache parameters are discussed:

- *Cache size:*

  How much space should be used for cache?

- *Line size:*

  How many information should be transferred between cache and main memory during one cache miss?

- *Set size:*

  What is the group size in cache to hold information?

- *Cache partitioning:*

  Should cache permanently be divided according to different usages (e.g. data vs instruction)?

2.3.1.1 Cache Size

Cache size is the total space used to store information and the cache control logic. The question of how large a cache should be is very difficult to answer. Since one of the main optimization issues of cache design is to achieve high cache hit ratio, the cache size should be as large as possible. If the cache size is large, more information can be stored in cache with less replacement; hence system performance is improved, provided that others factors are unchanged.

However, there are factors that limit the possible cache size. The first one is economical. The initial purpose of introducing cache in computer design is a cost-effective solution to bridge the access time delay gap between the fast CPU and the slow main memory. Since cache is more expensive than main memory, cache size should be as small as possible (in order to reduce the cost). If cache were of similar size as main memory, there is no need to have this extra memory level — cache.

The second factor is the possible VLSI chip area and/or board area available. There is a limit of how many things can possibly be implemented on a single chip or a board. The additional area that larger cache occupies might be used to implement other functional units. The benefit obtained from these functional units might (or might not) be greater than that from cache. The decision of using limited area for cache or other purposes needs to be made by cost-return evaluation.

Finally, larger cache needs more power and cooling, which increases the cost (economic reason) and the area required. Moreover, since larger fan-in and fan-out of the gates in a high-capacity cache introduce longer rise times, large caches tend to have larger cache access time than the small ones. This is true even when it is built with the same integrated circuit technology and is put into the same place on a chip or circuit board.

Since the performance improvement of cache levels off when the cache size exceeds a certain value, the general rule is to choose a cache size that is slightly beyond this value (This value depends both on the architecture and the application domains and is usually found by simulation).

2.3.1.2 Line Size

A **line (block)** is defined as the basic unit of information, consisting of one or more information elements, transferred between cache and main memory. It is also the amount of information (in words) which might be associated with a cache tag. Usually, the line size should not be wider than the bandwidth of the main memory because it is often desirable to transmit an entire line in one memory cycle.

Simulations from current cache research on line size selection suggests that as the line size increases, the cache hit ratio first increases. This is because more information in the current working set is fetched for each cache miss. However, as the line size becomes larger, the probability of

reusing information in a cache line that has already been replaced increases. This is due to the smaller number of cache lines available and more information stored per line. Consequently, the cache hit ratio decreases after the line size is larger than a certain number [Smi82] [PoA83].

Line size also affects the penalty for cache miss. Since the penalty for cache miss is the overhead time for each cache miss plus the product of the line size and the transfer time per byte between the cache and the main memory, longer line size implies a higher penalty for each cache miss. Usually, the overhead time is much greater than the byte transfer time (by a factor of 10 to 20). This is true no matter how efficient the time overlapping between the memory access and the CPU execution is. Hence, if the line size is larger, the time saved for fetching a fixed amount of information increases (due to the same large overhead).

On the other hand, larger line size also implies longer tie-ups at the memory interface within a short period of time. This burst mode of data transfer is definitely undesirable in multiprocessor systems. Since one processor might lock out other processors from using the shared memory while it handles a cache miss, the whole system might be slowed down.

There have been attempts to find out the relationship between the cache miss ratio and the line size [PoA83] [Smi87a]. However, all formula proposed are found by curve fitting on experimental results which they obtained from their simulations on some limitied number of benchmark programs. No application of these formula has ever been made. Currently, the line size that many VLSI processors used is about four.

While the above guideline of cache line size selection has been practiced for a long time, it does not necessarily gives a better execution time. *When the line size changes, the time needed to transfer a line between cache and main memory changes.* Hence, performance of a cache with smaller line size and higher cache miss ratio might be better than that with large line size and lower cache miss ratio. It is surprising that many cache research papers published on this subject overlook this fact and use cache miss ratio as the only optimizing parameter.

2.3.1.3 Set Size

The **associativity, or set size**, of a cache is the number of information elements (cache lines) in a cache set. All lines within the same cache set map to the same domain of lines in main memory and they are associatively searched for information. **Direct mapped cache** is a cache organization with cache set size equal to one. When the cache set size is equal to the cache size (in lines), it is said to be **fully associative mapped cache**. Any cache with cache set size in between these two cases is called **s-set associative cache**, where *s* is the cache set size.

Current research suggests that when the associativity of a fixed size cache increases, the cache hit ratio increases rapidly. Afterward, the increase in cache hit ratio levels off. Two way associativity — a set size of two elements — is significantly better than direct mapping. Four-way is better yet, although only slightly; further increases have little extra effect [Smi78c]. Beyond eight-way associativity, the cache hit ratio is unlikely to have any further increase. Typical choice of set size is four or eight.

There is a bad side-effect of increasing cache associativity. More components and connections of cache logic are needed to read out and to compare address. Hence, the cache access time is larger for higher cache associativity. This is more important for on-chip cache, where the chip area is very expensive.

As an example, suppose the associativity increases from one to two. An additional levels of control logic increases the cache access time. Consequently, for large caches, where the cache miss ratio is already low, the faster access time of a direct-mapped cache might give better performance than that of a set-associative cache. Conversely, for smaller caches, where delays from cache misses dominate, the set-associative cache is preferable.

Again, the side effects of having larger set size — the additional area occupied by the cache control logic and the longer cache access time — are often overlooked. In the past few years, this has been improved and cache designers start to consider these side-effects in the cache set size selection. This explains why many current high performance VLSI processors employ direct-mapped cache instead of set-associative cache.

Although it is commonly believed that an increase in cache set size *always* increases system performance, it is proved to be incorrect. Direct mapped cache might sometimes have a better cache hit ratio than fully

associative cache [SmG83] [SmG85]. Moreover, if total memory access time is used as the optimizing parameter (instead of the cache hit ratio), we found that cache with smaller set size often give better system performance than that with larger set size.

2.3.2 Cache Partitioning

The rationale of cache usage is based on the locality properties of references. Different types of references (instructions and data) have different reference patterns (i.e., a different locality set). For example, the reference pattern for instructions is very regular and predictable since it is sequential or iterative in nature. Data references, on the other hand, are typically more randomly accessed and are hard to predict purely in hardware. Hence, it is suggested that cache should be partitioned into disjoint sets according to the type of references. This is because locality of reference for each partition can be improved, resulting in better cache performance.

A good example is to have separate data and instruction caches [Rad83]. There are several advantages for this design. First, reference bandwidth of partitioned caches is doubled since two requests can now be serviced simultaneously. Reference conflicts between simultaneous instruction fetches and data reads and writes can be reduced.

Second, it is easier for each cache partition to capture its own working set of references. The reference pattern for instruction is more regular than that for a mix of instructions and data.

Finally, cache access time in each cache partition might be reduced by a fraction of a nanosecond to several nanoseconds. In a single cache system, it is difficult to place a cache immediately adjacent to all control logic that accesses it. However, in a partitioned cache, since the size of each partition is smaller (relative to the entire cache), each partition can be placed in the physical location closer to the control logic that accesses it.

On the other hand, there are some tradeoffs in cache partitioning. Once cache space is partitioned according to some reference usage, it is very difficult for different partitions to share the same cache space. This might create problem of inefficent use of cache space — one partition does not have enough space to store its information while the other partitions might only be slightly occupied. This problem does not exist in single cache system

because the cache space can be used for either instruction or data. Furthermore, the problem of how a given cache space is divided into partitions is difficult to answer because the demand of cache space for each program is different. All these problems are due to the inflexibility of "hardware partitioning".

Currently, partitioned cache (separate data and instruction cache) seems to be more popular than single cache. When there is insufficient space for all partitions (e.g., in on-chip cache design), instruction cache is always chosen to be implemented.

2.3.3 Fetch Algorithms

Fetching policy is a mechanism to decide when information should be moved from main memory to cache. There are two types of fetching policies — demand-fetching or prefetch.

Demand-fetch is the policy in which cache lines are brought into the cache only when they are needed and are found to be absent from the cache [Spa74] [Smi82]. Therefore, the processor might become idle until requested data/instructions are received from main memory. In high speed computer systems, this might become one of the performance bottlenecks. The main benefit of demand-fetch is that any fetched line is referenced at least one time before it is replaced. This is in contrast with prefetch, where a fetched line might be replaced before it is ever referenced.

In **prefetching**, references may be brought into the cache before they are actually needed [Smi78a] [Smi78b]. Memory cycles that would otherwise be idle are used to copy data into the cache. While program execution time might be reduced when cache prefetch is used, the cache hit ratio can be increased. Improvement in performance usually comes from the overlapping of the memory access time and the CPU execution.

There are two approaches to prefetching: **static prefetching** (which is done at compiled-time), and **dynamic prefetching** (which is done at run time).

Prefetching has great potential to improve cache performance. The key difficulty is deciding what to prefetch and when. For dynamic prefetching, the usual prefetching policy is to prefetch cache line $i+1$ when cache line i is referenced and not in cache, i.e. one line lookahead. This is the most common prefetch scheme because, for a long time, people believe that in

cache memories, due to the need for fast hardware implementation, the only possible line to prefetch is the immediately sequential one [Smi82].

However, this causes a serious problem (especially where cache size is small) in that severe cache pollution often results. Information may be prefetched into the cache, replacing some cache line(s) that are referenced later with prefetched information which is never referenced.

To remedy this situation, the *working set restoration* approach makes a record of the contents of the cache whenever the execution of a process is interrupted. When the process is restarted, the cache contents (before interruption) are restored, or even better, only its most recently used half before the interruption. The problem with this approach is the large overhead involved and the fact that not all the things restored are used again.

For static prefetching, prefetching can be made "smart" — so that all information prefetched eventually will be used. Cache pollution is minimized (but not eliminated) with this type of prefetching. This improvement may be achieved without a significant increase in complexity of the cache hardware. Further, since the prefetching operations are scheduled into times when no memory-to-cache traffic is anticipated, there is not likely to be any interference with normal fetching. However, new compiler technology is needed to implement this "smart" prefetching.

If prefetch mechanism is not handled properly, both the cache miss ratio and the total program execution time actually increase. This is because a prefetch instruction which specifies the transfer of large amounts of information would run the substantial risk of polluting the cache with information that would not be used at all. On the other hand, if only a small amount of information were prefetched, the overhead of the prefetch might exceed the value of the savings. In current cache design, either demand fetch or dynamic prefetch is usually used.

One interesting implementation of fetching algorithm is **fetch bypass or loadthrough**. When a cache miss occurs, the desired bytes can be passed directly from the main memory to the instruction fetch unit, bypassing the cache. In this case, the cache is loaded, either simultaneously with the fetch bypass or after the bypass occurs. This method is used in 470V/7, 470V/8 and IBM 3033 [Smi82].

2.3.4 Replacement Policies

Replacement policy is defined as the set of rules by which the choice of cache line to be replaced is made when the cache is full and a new line is to be fetched from the main memory into the cache. In current cache design, there is a constraint on the replacement policy that needs to be obeyed: the cache replacement policy must be implemented in hardware and must be executed very fast so that there is no negative effect to the processor speed.

Hardware-implemented replacement policies such as LRU (least recently used), random replacement, FIFO (first-in first-out) etc. are commonly used in current cache design. These polices can be classified as implementing one of two general models: a history-based replacement model or a probabilistic replacement model. For the history-based model, LRU is an example; random replacement is an example of the probabilistic replacement model.

In addition to these two schemes (LRU and random), there are other replacement policies like MRU changes [SoR88], FIFO [Bel66], etc. However, LRU and random replacement policies are the two most common schemes with the best average performance. All others either have an average worse performance or are derived from these two policies. Hence, only LRU and random replacement polocoes are discussed here.

2.3.4.1 Least Recently-Used (LRU) Policy

The **least recently-used (LRU) policy** for cache replacement chooses for replacement that line in cache which has not been referenced for the longest period of time [Bel66] [Spi76]. The LRU stack consists of a list of k cache lines referenced by a program in order of most recent usage, where k is the cache set size. Cache lines that are referenced most recently should be placed near the top of the LRU stack, in the hopes that these lines will have the highest probability to be referenced.

Instead of using "pure" LRU, it is far more common that an approximation to LRU is implemented using a one-bit time stamp [PeS85] to reduce the control logic complexity. It is unlikely that such an approximation to LRU would perform as well as LRU, and very unlikely that it would perform better.

LRU does not always have good performance throughout an entire reference program. Since reference patterns tend to change from one region

of code to the next, working sets of references change in time. Frequently, this change is gradual, cache line by cache line; occasionally, the working set of references of a real program is completely disrupted as the program begins a new phase of execution. Under this condition, cache performance using LRU is very bad because the overlap of two working sets of references is usually very small. Information that is good for one working set of references is likely bad for other working sets.

For example, suppose a program makes a pass through successive elements of a large, multi-cache line array. When the first array element in a given cache line is referenced, the cache line enters the working set of references and remains there until the last element is referenced. This process continues for each cache line of the array. In this way, the working set of references changes slowly in time. However, after the pass is completed, the program may begin an entirely new function, using a new working set of references which may overlap little, if at all, with the old. Such phase transactions naturally induce clusters of cache misses, since an entirely new working set of references must be acquired on demand but the working sets in LRU stack model change by only a single line at a time.

The performance of LRU is even worse when the cache size is smaller than the size of the current working set of references that are referenced repeatly (e.g. references within a loop with size larger than the cache size). Under this situation, cache lines that are replaced are those that are going to be referenced in the near future.

Consequently, LRU only models reference behavior which is within a single phase of execution.

2.3.4.2 Random Replacement

In the **random replacement policy**, the fundamental assumption is that references occur at random, i.e., evenly distributed over the range of all program lines and each line referenced is totally unrelated to other lines [Bel66]. This is clearly opposed to what the principle of locality of references suggests about program reference behavior. The main argument for choosing this replacement scheme is that *it is better to assume nothing than to assume something that is always wrong.*

Under this assumption, historical information is irrelevant, and the use of any specific replacement rule does not ensure any relative advantage.

Therefore, we might as well choose a simple, random replacement scheme in building the probabilistic model. This scheme chooses the cache line to be replaced at random over the range of all lines in a cache set.

The average performance of LRU and random replacement are about the same and are better than the performance of other algorithms. However, under certain program structures (e.g. loops with size larger than the cache size), it is shown that random replacement always performs better than the LRU [SmG83] [SmG85] (discussed in Chapter 4).

2.3.5 Update Policies

A **update policy** is the set of rules whereby it is determined whether a datum being stored should be placed in cache or directly into memory and, if placed in cache, when and how the main memory cell should be updated. Since conventional wisdom marks instructions as read-only (typically, self modifying code is not written), the update policy applies only to stores of data.

If stores do pass through the cache, there are at least two basic strategies for managing them: **write through** the cache to main memory or **copy back** data from the cache to main memory only when the cache slot must be re-used. Write through transmits modified data immediately to main memory; thus, all write instructions result in data being transmitted to main memory. Copy back transmits the entire modified cache line to main memory when a cache miss occurs and that cache line is selected for replacement. Since it is only necessary to copy back a cache entry whose value has been changed (so that it no longer matches the value in the location backing it in main memory), a "dirty bit" is often used to mark such cache lines.

For write through, the cache and backing main memory are always **consistent** — corresponding locations always hold the same values. In a multiprocessing environment, where the cache is used in shared memory systems, write through is a simple way of insuring cache consistency — different copies of the same information in local cache of different processing elements have the same value. Also, its implementation is simple, merely forcing a write to main memory for every store instruction. However, it has the disadvantage of making unnecessary memory update.

In a multiprocessor (multi-cache) system, if the write through is accomplished without blocking the processor and pending completion of the write into memory, it is possible that the processor would signal another processor to read the value from memory, causing that read request to reach the memory before the write has completed (since it may take a different path through the interconnection network). The alternative, which is blocking the processor until each write has completed, greatly impedes performance in general. In most implementations using copy back, longer delay is experienced when a cache miss occurs, since the value originally in the cache must be written back to main memory before it can be replaced by the value just referenced. Also, extra logic is needed to implement "dirty bits." Write back, however, may give a lower cache miss ratio than is achieved using write through [Smi82].

It has been shown that each of these two memory update policies has better performance over the other under different conditions [Smi82]. The choice of update policies depends on the application domain and the architectural design of cache.

Ideally, a system would incorporate both memory update policies (without excessive overhead) and would optimally choose the update policy to be used for each data write in a reference program. However, because of the lack of global knowledge of future reference behavior in traditional cache design, it is impossible for the hardware to decide when and how to switch the memory update policy. Consequently, this policy is not considered in any current cache design.

2.3.6 Optimization Issues

Since the first implementation of cache in 1966, the *only primary optimization parameter* in most cache designs has been the **cache hit ratio** [PoA83] [HwB84]. It is defined as the chance of finding the referenced information in cache without causing any information transfer between cache and main memory. Although other performance issues like bus traffic are also used, they are *far less popular.*

It is believed that cache hit ratio can reflect accurately and directly how well the cache contributes to the system performance. It is assumed that the performance of a system is always better with a higher cache hit ratio than with a lower cache hit ratio.

However, this performance measurement using cache hit ratio is misleading. *A system with a high cache hit ratio might not imply a smaller total execution time than the one with a low cache hit ratio if the other cache parameters in these two systems are different.* A cache miss only indicates that the reference information is not in cache. It does not indicate the *penalty cost* of a cache miss, which is related to other parameters such as the line size.

An example might be helpful to clarify this argument. Suppose a certain cache design has a cache hit ratio of 0.8, cache line size 1, and the penalty for cache miss is 10; hence, the average access time of each memory reference is 2.8. By increasing the line size of this cache design from 1 to 4, the cache hit ratio increases to 0.9 and the penalty for cache miss increases to 30; hence, the average access time of each memory reference is 3.9. If cache hit ratio is the primary optimizing parameter, the latter design is better. On the other hand, if execution time is the primary optimizing parameter, the former cache design gives a better performance.

This issue will be discussed in detail in Chapter 6. What can be said here is that it is better to use execution time as a performance measure than the cache hit ratio.

2.4. Software Cache Management

In the past few years, cache research on uniprocessor has become relatively inactive. Cache design for uniprocessors is considered as a necessary but tedious and unchallenging routine in computer design. Peforming cache simulations to select an optimal set of cache parameters and management schemes is very time consuming. However, the performance improvement obtained from these simulations is very small — usually a few percent. Furthermore, this improvement can only be obtained within a particular domain. Those fundamental cache design problems that existed in the 1970's still remain unsolved.

This has resulted in two main research directions in cache design. The first direction believes that major design issues in uniprocessor cache have been completely covered and there is little hope (if any) to get any great breakthrough in improving cache performance. Current design guidelines for uniprocessor cache can be assumed to be *good enough* for most purposes. Hence, it is better to shift research efforts to other *more interesting cache*

related topics such as multiprocessor cache design [LeY87] [Vei87] or I/O transfer.

The second research direction in cache design is to include program flow analysis into current cache management schemes, hoping that this can solve some of the "basic hard problems" that most cache designs always face. Research in this software driven cache management includes some of the following issues:

- Cache prefetch

  Reference sequences obtained from program flow analysis is used to help decide what should be prefetched [PaG83] [Bre87].

- Program transformation

  Programs are restructured (or transformed) by the compiler to improve localities of references [Tha81] [Mac83] [Mac87] [Abu79].

- Cache consistency

  Data consistency in multiprocessors is maintained by explicit software control [ChS86] [Vei85] [Vei86].

Since the main focus of this thesis is to obtain the best performance from a fixed cache structure and a given reference program (i.e. no program transformation) in an uniprocessor environment, only the first issue — cache prefetch — is discussed in a separate sub-section. The other two issues are discussed briefly below.

Program transformation can improve cache performance by matching cache organizations (e.g. cache size) to the program reference pattern. There are two main aspects in this research: loop fitting [Abu79] [Tha81] and address re-mapping [Mac83] [Mac87]. It is observed that if the entire loop can be fitted into the cache, the cache performance can be greatly improved because no main memory fetch is necessary during the loop iteration (discussed more detail in Chapter 4). Hence, program transformation can be performed so that a loop with size larger than the cache size can be broken down into several smaller loops, each of which can be fitted into the cache entirely.

The second aspect of program transformation to improve cache performance is to rearrange the layout (address mapping) of information in main memory. Information that is referenced within a short period might be stored in many different cache lines. By re-packing this referenced information into fewer lines, the number of lines that needs to be fetched

and that stores in cache is reduced; hence the performance is improved. Potential memory conflicts caused by simultaneous accesses of the same memory module can also be avoided by storing this information into different memory modules.

In multiprocessing environment, an additional cache design issue is the data consistency problem — to force different copies of the same data (i.e. under the same name or label) in different processor elements or memory modules to have the same value [DuB83] [LeY87]. The cache consistency problem is a major bottleneck to multiprocessor cache design. This is because the overhead involved in maintaining consistent data might sometimes be larger than the benefit gained by storing the information in cache and the hardware needed to maintain data consistency is very complex. Only data references have this consistency problem because instructions are almost assumed not to be self-modifying.

Given a program, flow analysis is performed to find out its data dependencies and the parallel execution mode of the program. Then, certain information might be designated as non-cacheable, and is accessed only from main memory. Such items are usually semaphores and perhaps data structures such as the job queue. Access to shared writeable cache data is possible only within critical sections, protected by non-cacheable semaphores. Within critical sections, the code is responsible for restoring all modified items to main memory before releasing the lock. An example is the S-1 multiprocessor system built at Lawrence Livermore Laboratory.

Simulation results show that a large potential improvement in cache performance might be obtained by these software cache management schemes. However, these schemes are not popular because the analysis involved is very complex and most of them are not complete — only some reference structures (e.g. array indexing) are discussed. Moreover, since they are still based on most of the basic assumptions of cache designs discussed in Section 2.2.2, large potential cache improvement obtained by fine-tuning these "invalid" assumptions cannot be obtained by any of these approaches.

2.4.1 Software-managed Cache Prefetch

As the memory delay gap between the CPU and main memory increases, sequential cache prefetch [Jos70] [BaS76] [Smi78a] [Smi78b] is no longer sufficient to reduce the total memory access time. This is particular true in VLSI environment, where the on-chip cache size is small and the ratio of off-chip to on-chip access time is very high [Hil83] [HiS84]. Small cache size and inaccurate line prefetch result in serious cache pollution. in order to prefetch "useful" information, attempts are made to incorporate program flow analysis into cache prefetch schemes [PaG83] [Bre87].

In [PaG83], the idea of a remote program counter (RPC) was proposed to predict the address of the next instruction. This RPC resides in the instruction cache and consists of a register, adder, and multiplexer. It utilizes the predictable nature of PC address to prefetch the next instruction. Since the subsequent PC value is obtained by incrementing the present value by a small offset, the RPC predicts the new PC value and compares it to the real PC while fetching the next predicted new PC value. If there is no match, another cycle is needed to check if the next instruction corresponding to the real PC is in cache. One advantage of the RPC is that the time available to read the cache data and address tag (critical path of cache access) is nearly double. For successive sequential references, this approach obviously improves cache performance. However, for conditional branches and interrupts, RPC loses its effectiveness.

In [Bre87], Brent proposed the use of program structure to achieve cache prefetch in cache memories. In his approach, a program skeleton, which describes the structure of the source code as well as the low-level memory accessing behavior is developed. From this program skeleton, transformations are developed to create a machine-specific cache memory prefetching control program, called the prefetch skeleton (PFS). A prefetching unit (PFU) is developed as a simple in-cache processor. This PFU executes the PFS and generates cache line prefetch requests ahead of CPU demand requests. This approach can be considered as a slightly improved version of the remote program counter approach. Again, this approach only works for instruction fetch because of the predictable nature of instruction references. Since data references are more randomly accessed, the improvement of data cache performance using this approach is very little (if any).

In all these schemes, cache prefetch is only issued one or two instructions ahead of the instruction currently being executed. This is certainly not a good strategy because the bus might be busy when the cache prefetch starts and the prefetch process might not be finished in time to supply data to the processor. However, if the cache is explicitly managed by the compiler, cache prefetch can be initialized much earlier, resulting in a lower chance of being blocked by the busy bus traffic.

2.5 Conclusions

In this chapter, the rationale of cache usages — program reference behavior, basic assumptions and optimization issues — and various cache design choice selections were discussed. Design guidelines as well as the motivations behind were also given.

The principle of localities of references describes reference behavior of programs and suggests what information should be kept in cache within a working set. However, this principle fails to apply when the working set of references either changes or is larger than the cache size. It is used only because the reference sequence order is *assumed to be unknown.* However, with program flow analysis by the compiler, this reference order is not difficult to obtain (although some compile-time uncertainties may exist).

Basic assumptions and optimization issues used by most current cache designs have also been discussed. It is found that most of these assumptions and optimization issues are either not accurate enough or obsolete given the development in compiler technology. Future references are not difficult to obtain by the compiler; and the "cache through" idea might not be necessary. In addition, cache hit ratio is the wrong measure of system performance. All these facts limit the improvement of cache performance.

For replacement policies, it is found that the two most commonly used hardware-implemented cache policies are based on either the history of execution or a probabilistic model. Each of these policies is tuned to a reference pattern obtained by a "guess" using no knowledge of the program structure; hence, whenever the data/instruction reference pattern of a real program being executed happens to approximate the reference pattern from which the cache policy was derived, good performance is achieved. For example, if a program is in the middle of a region of code and strong localities of time (temporal locality) and/or space (spatial locality) are

present, then cache policies based on a historical model may have good performance. However, as the program passes from one region of code to the next, the same cache policy may evidence the worst possible performance. It is because the pattern assumed is independent of knowledge of program structure. Hence, the performance of current hardware-implemented cache policies is typically far from optimal. A similar situation occurs relevant to fetch policy and write policy.

For fetch policies, it appears that "smarter" prefetching can increase the cache hit ratio significantly. The main complication is that the impact of cache pollution must be taken into consideration and, even without prefetching, this problem may make it profitable to reference directly from memory as though there were no cache (thereby avoiding pollution of the cache). Since pollution is caused by single-event perturbations in the referencing structure, no history-based model (e.g., OPLA [BaS76]) is effective: when the event becomes history, it has already polluted the cache. Furthermore, the time between the start of cache prefetch and the execution of prefetched data might be too short for the cache prefetch to complete.

In update policy, the choice of write through or write back cannot generally be decided in favor of one or the other: there are situations in which either is better than the other. To obtain optimal performance, one needs a software-driven technique for chosing the best write policy for each write operation in the program.

The sources of major performance improvement include: different regions of code have different locality sets which have little (if any) overlap, branches and subroutine calls also skew localities in a certain way, and different applications have different kinds of locality (spatial versus temporal). These cannot be approached as hardware design problems, since, as discussed above, hardware techniques are inordinately expensive per unit performance improvement: hybrid hardware cache policies (e.g., [Bab82]) are expensive to implement, but their performance in fundamentally limited by the total lack of knowledge about future program behavior. Global information about data/control flow should be incorporated into cache policies.

A relatively minor additional point is that the cost variations for different types of memory references cannot easily be incorporated into the hardware-implemented schemes. For example, in parallel processing systems, referencing from different memory locations may imply different

costs, since memory space may be partly local and partly global (within a single address space). The difference between referencing a variable stored in global memory and a variable stored in local memory may be a factor 10 or more — any reasonable cache policy needs to incorporate understanding of these weights.

Recent advances in compiler flow-analysis techniques [AlB86] [BuC86] [Die87] make global control/data flow analysis of programs practical. Hence, it is now possible to improve cache performance using predictions of program behavior based on global control/data flow of programs. This technology provides the ability to obtain high-probability reference strings at compile time by simply looking ahead in the program's flow structure. Given a reference string at compile time, both demand-fetch and prefetch cache policies can be "fine-tuned" to the actual references which the program will make.

The flexibility and power of a software-implemented policy, as well as the ability to obtain and use global information about program behavior, make a software policy far more promising. Hence, we propose to migrate hardware-implemented cache policies into software and to use the compiler to improve, and in some cases make optimal, the runtime performance of an architecturally very simple cache.

CHAPTER III
COMPILER-DRIVEN CACHE MANAGEMENT USING MACHINE LEVEL STATE TRANSITION (MAST)

3.1 Introduction

In Chapter 2, it was suggested that all common hardware cache policies are based on either historical or probabilistic models. Hence, each hardware management scheme performs better than the others under certain memory instruction/data reference patterns: no purely hardware "fix" can be made to improve performance because it is not feasible to put all these hardware policies together and to dynamically change management schemes as the memory referencing pattern changes.

A natural alternative is to improve performance by *modifying the structure of programs*, at compile time, to match the ideal reference patterns for the hardware management scheme in use. It is, however, impossible to transform arbitrary code into a perfect match for a *single* hardware management scheme. For this reason, we propose to allow the compiler to explicitly *control the operation of the cache for each reference.*

Detailed global control/data flow analysis of programs enables us to know more about the order of instruction execution and about the data used or defined by each instruction [AlB86] [AhS86] [BuC86] [Die87]. In effect, this analysis can determine either the exact reference sequence or a set of possible reference sequences and their associated probabilities of occurrence at runtime. This makes a software cache management scheme feasible — if this information was not obtained automatically (using compiler analysis), very few users would be willing or able to explicitly state cache control for each reference.

The optimal control of a cache using compile-time information does not, however, require an increase in the complexity of the cache hardware. Rather, this control simplifies it, since the hardware no longer needs to make decisions, but merely implements them on command. If a particular reference is "marked" (by the compiler) as being cached in a certain way, it

is of no great concern to the hardware that the previous reference was "marked" to be treated differently — as far as the hardware is concerned, the cache management scheme is to do what it is explicitly told. In effect, ordinary general-purpose registers within a processor have long been managed in exactly this way: cache "registers" (entries) are not really so different.

As in performing good register allocation, the overhead imposed is that complex compiler technology must be designed and implemented. But, aside from improving performance in much the same way as registers do, this overhead is justified by the simplification of the hardware relative to achieving a given cache hit ratio — the VLSI area saved, particularly in an on-chip cache, is priceless. Furthermore, as is discussed in Chapter 5, simple, linear-time complexity versions of this model, which can regain a large percent of the performance loss over current cache management schemes, can easily be derived from this model. The hardware modification required is very small.

In the following sections, a compiler-driven cache management model based on the **MA**chine level **S**tate **T**ransition, called the **MAST Model**, is described as an alternative cache management scheme. The basic idea of this model is to analyze global control/data flow of the program and to have the compiler *explicitly* manage cache activity based on this information. Toward this, the global control/data flow graph obtained from the analysis of a program is expanded to include all feasible cache contents at each **cache stage** (defined below) in the graph. The cost for each *transition* of cache contents from one **cache state** (defined below) to the next is then placed in the graph as the weight of the edge linking the two cache states. An algorithm based on shortest path is executed to obtain an *optimal*[1] cache policy for each cache transition. This information is then used by the compiler to generate code which explicitly controls the cache, either through "cache instructions" (treated much like coprocessor instructions to be executed by the cache engine) or as tags on each instruction.

Throughout the analysis in this chapter, it is assumed that the reference pattern is *precisely known* at compile time. That is, we ignore the fact that some references will be ambiguous: for example, a pointer might be known to refer to one of two different memory cells, but the compiler may

[1] The word "optimal" here is defined in terms of execution time.

not know which. These complications, as well as code structures including branches (as opposed to the branchless reference strings in this chapter), will be discussed in Chapter 4.

The organization of this chapter is as follows. Section 3.2 states the assumptions made in the MAST Model. In Section 3.3, notations and definitions of terms used in the analysis are introduced. Section 3.4 describes the graph formulation of the MAST model of programs. In Section 3.5, the algorithm implementing the compiler-driven cache policy is described. The MAST model in different cache organizations is discussed in Section 3.6. Finally, the chapter concludes in Section 3.7.

3.2 Assumptions of the MAST Model

There are several assumptions made in the description of the MAST model. These assumptions are not crucial to the model, but rather serve to make analysis and comparison with other alternatives more manageable for this presentation. They are *not the constraints of the MAST model.*

As will be seen in the next two chapters, most of these assumptions are removed and additional performance or simplification of the MAST model can be obtained.

Assumptions of the MAST model made in this chapter are:

- Fully associative cache organization is assumed in the description of the MAST model here. As will be discussed at the end of this chapter, other cache organizations such as direct mapping and set associative can be transformed into sets of subproblems, each of which has fully associative cache organization and a smaller cache size. In fact, cache organizations with smaller set associativity can help reduce the computational complexity of the MAST model and the hardware complexity of cache control.
- There is no restructuring of program control flow nor rearrangement of the data/instruction storage patterns. Of course, program restructuring and rearrangement of data/instructions storage patterns by an optimizing compiler can improve cache performance [Abu79] [Mac83] [Mac87]. For example, if some cache sets are used more heavily than the others, an optimizing compiler can always re-define the name-address mapping so the workload can be distributed more equally among various cache sets. However, this is outside the scope of this

thesis.

- The reference string is known at compile time (i.e. branch-less code, with completely unambiguous data references, is assumed). Applications of the MAST model to other program structures (e.g. loops and branches) and ambiguous references are discussed in Chapter 4.

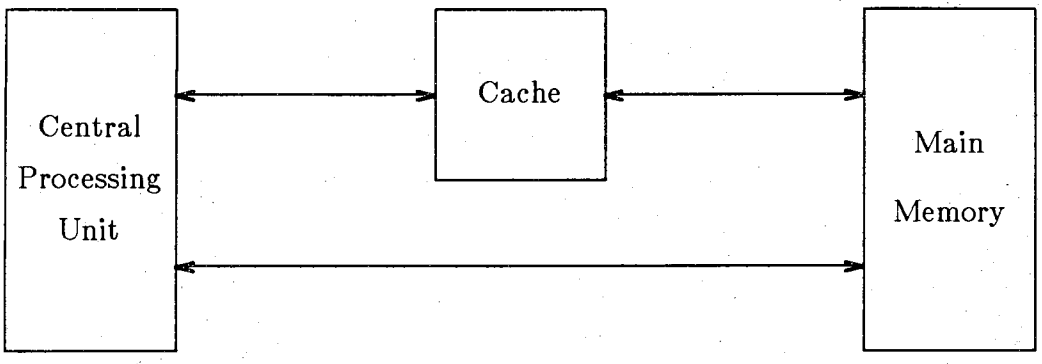


Figure 3.1: Model of Processor/Cache/Memory Interface

- The central processing unit has the capability of directly accessing the main memory without going through the cache (as is shown in Figure 3.1). Sometimes, it is more economical to reference an item directly from the main memory than to transfer the entire line into the cache first before it is referenced. The large overhead involved in cthe line transfer may not be justified by the infrequent use of that line. In this case, direct reference from the main memory is preferred. This option of cache bypass — referencing directly from main memory without going through cache — is actually not an necessary assumption. In fact, it can be removed and the model still works with lower computational complexity (fewer possible ways of changing cache configuration per reference). The only reason we make this assumption is that a large performance loss in current cache management schemes can easily be regained with cache bypass, which is very easy to be

implemented.

- When the information is referenced directly from main memory, that information will not be placed in cache. This is in contrast to current cache management schemes where a copy of the datum currently being referenced needs to be in cache. The MAST model provides a convenient treatment of different write policies: write through is modeled by using the direct-to-memory path, whereas write back requests travel through the cache. This is the key idea in several "intelligent" memory management schemes (e.g. a unified registers/cache management scheme discussed in Chapter 7) rather than a necessary constraint of the model.
- Memory hierarchy levels, except the interface between the cache and main memory, are transparent (or ignored) to the MAST model. Although this is not the optimal way to manage the memory hierarchy, this assumption simplify the model analysis. Moreover, a simple and intelligent model, the **unified memory management model**, will be proposed in Chapter 7 to manage registers and cache simultaneously and efficiently.

3.3 Notation and Definitions

Before describing the graph formulation of program, some definitions of notations and terms are needed:

- The reference string is denoted as $\omega = r_1, r_2, \ldots, r_n$, where n is the total number of references.
- Let $|M|$ be the set of *distinct* cache lines referenced in a program. It is found that the ratio of (number of references made)/(number of distinct lines used) (i.e. $n/|M|$) is very large; an factor of 100 or 1000 is not unusual.
- Let K be the cache size (in lines). It is assumed that $K \leq |M|$. Here, the general case of program size larger than the cache size is studied. The case where the program size is smaller than the cache size is not considered because the whole program can be placed in cache and any reasonable cache management schemes can perform very well.
- Let S_i represents the subset of M in cache after the reference r_i has been completed. For all i, $0 \leq |S_i| \leq K$ and $S_0 = \varnothing$.

3.4 Graph Formulation of the MAST Model

In this section, the compiler-driven cache management model using machine level state transition (MAST) for finding and controlling all cache placement/replacement activities is described. In this model, a reference program (more precisely program control/data flow graph) is obtained from a given program. This global control/data flow graph obtained is then expanded to include all possible cache contents at each cache stage. The graph is constructed in such a way that all possible complete sequences of feasible cache state transitions from the initial cache state to the final cache state are included in the graph. Each of these paths from the initial cache state (defined below) to the final cache state represents one possible complete sequence of feasible cache state transitions for executing the given memory reference string ω.

A simple algorithm is then used to find the path with lowest cache transition cost (i.e., the shortest path). After the optimal set of cache transitions has been found, the cache control is embedded in the code generated by the compiler (either as explicit cache prefetching instruction or as a tag included in each instruction).

Assuming that the memory reference string is known at compile time, this technique results in provably optimal cache performance (in terms of execution time). The optimal of use of the cache hardware is insured for any cache hardware design (within the bounds given above) and for any transition cost function. This ability to use arbitrary cost functions makes the MAST Model particularly attractive in control of multiprocessor caches, where hardware-implemented cache management schemes are typically unable to use the fact that different costs are associated with accesses of different memory locations (local or global). The optimality of the MAST model depends only on the reference string, and the cost function, being known at compile time.

In the graph formulation of the MAST model, there are five phases:

- reference program formulation (i.e. reference sequence construction for a given program),
- cache state construction (i.e. node construction at a particular time instant in the graph),
- cache stage construction (i.e. node construction for the whole graph),

- cache arc construction (i.e. arc construction in the graph), and
- cache arc cost assignment (i.e. weight assignment to each edge in the graph).

In the MAST model, the corresponding line number of each reference address is used. This simplifies the analysis in the MAST model, with no ill effects.

Before describing the graph formulation of the MAST model of a program, the important concept of live range of references in cache is introduced. Surprisingly, this live range concept is completely missing in current cache design research.

3.4.1 Live Range of References in Cache

An important concept in register allocation is reference liveness. In the literature on caches, one finds liveness mentioned only as a technique for reducing overhead in analysis of program traces [McD88]. It is also tempting to think of liveness of *addresses* — rather than liveness of values stored in them. Instead of using addresses or names, the definition of live range of an item to be cached should be in terms of values — exactly as in register allocation.

To give a formal definition of live range of a reference in cache, terms from conventional, sequential, compiler analysis [AhU77] [AhS86] [Die87] are needed. These terms are **use**, **def**, **D-U chain** and **U-D chain**.

Definition 3.1: Def

A **def** (definition point) is an operation which temporarily binds a value α to a name β.

Definition 3.2: Use

A **use** is an operation obtaining the value α which is bound (at that point in execution) to a particular name β.

It is important to note that the above definitions specify that a use is really *a use of a def or set of defs and is not a use of a name.* In other word, a particular use, μ, of the value bound to the name β does not necessarily have any relationship to the value bound to β by a specific def; only defs which could be the last def of β executed before executing the use μ are significant.

The concepts of D-U and U-D chains embody the information directly relating defs and uses:

Definition 3.3: D-U Chain

The **D-U chain** of a particular def γ which establishes the binding name-value β-α consists of a set containing the def γ and all uses of the value bound to β where, for each use μ, there exists at least one control flow path from γ to μ such that no other defs of β are encountered along that flow path (γ reaches μ).

Definition 3.4: U-D Chain

The **U-D chain** of a particular use μ of the value bound to a name β consists of a set containing the use μ and all defs of β such that for each def γ, μ is in the D-U chain of γ.

There are two separate reasons that the U-D chain of a particular use might hold more than one definition. First, the use might follow a conditionally executed definition of the variable in question. Second, the use may refer to a variable whose identity is not precisely known at compile-time. In general, ambiguous aliasing makes it necessary to treat each def (or use) as a set of possible defs (or uses), one for each of the possibly-aliased names.

With this terminology, the concept of live range of a reference can be defined in terms of def and use:

Definition 3.5: Live Range of a Value

The **live range of a value** λ is defined as the set of instructions during which the value λ exists and may be referenced. In other words, it is the *D-U chain* of $\lambda \cup$ all instructions which, on some flow path, may be executed after λ's *def* and before the last *use* of λ on that flow path.

However, unlike registers, cache may hold instructions. For this reason, it is necessary to define the live range of an instruction:

Definition 3.6: Live Range of an Instruction

The **live range of an instruction** κ is defined as the set of instructions, including κ, which may be executed after the first execution of κ and before the last execution of κ on some flow path. Notice that for straight-line code the live range of an instruction κ is always the set $\{\kappa\}$, however, if κ is enclosed in a loop or multiple-caller subprogram the set may be greatly enlarged.

Although these definitions are not surprising, they have some surprising implications.

Perhaps the most dramatic of these is that a value which has become dead need not be stored back to main memory. Hence, suppose that the compiler is able to determine that a particular memory read operation of the value λ will be the last *use* of λ. If the value λ is cached, even if the cached value λ does not match the value which is stored at the corresponding address in main memory[2], the cache cell containing value λ need not be stored back to main memory. The compiler may simply inform the cache that this was the last reference to α and hence that the cache line which holds α is "empty" at completion of this *use* of α[3].

The benefit of having such "empty" cache lines is that instead of the basic line-replacement operation, only a simple placement is required to install a new line in cache. Of course, this also insures that no useful item was accidentally flushed from cache in this process. That is, when line replacement occurs, *this dead line in cache is the first one selected to be replaced.*

3.4.2 Reference Program Formulation

A **reference program** is the skeleton of a program which describes sequences of memory reference addresses and the corresponding reference mode — read or write — when a program executes. Only the memory reference addresses, mode of references, and the program control flow are stored in a reference program. The actual values stored in these memory location is not part of a reference program and is ignored. A **reference string** is a sequence of memory reference addresses and its corresponding reference modes (read or write) which a particular flow path makes when that path is executed. This is often called the reference sequence of a "program trace" [Ell85].

Values in a referenced program can either be instructions or data. There is no distinction made between these two types of references in the

---

[2] This occurs if a *def* of λ is executed when λ is in cache or when a *def* of λ creates the cache line and the cache has not yet stored λ back to main memory.

[3] In the interest of simplicity, this discussion has pretended that a cache line holds exactly one value — this restriction is easily removed, although more complex bookkeeping is required.

MAST model. Techniques and algorithms of the MAST model described in later sections can be applied for any type of references without any modification. Of course, if cache of some special use (e.g. data cache) is employed, only references of that use (e.g. data) need to be extracted from a given program.

In the first phase of the MAST model, the program flow graph and the reference sequences made for each flow path in the graph are determined at compile-time. Since it is analyzed at compile-time, the exact execution reference sequence might not be known. Instead, a limited number of "possible" reference sequences from different flow paths might result. For example, the final target of a 2-way branch is not known at compile-time, resulting in 2 possible reference sequences after the execution of a branch instruction.

In current optimizing compilers, the cost of this phase is almost "free" because program flow and data dependency analysis is a *necessary* step in most compiler optimization techniques [Ahu77] [AhS86] [AlB86] [BuC86] [Die87].

An example of the reference program formulation might help clarify this. Suppose a sample program segment is given in Figure 3.2, and the cache is only used for data references, the reference string corresponding to this program segment is shown in Table 3.1.

$$c = a + b$$
$$d = c * b$$

Figure 3.2: A Program Segment

Table 3.1: The Corresponding Data Reference String

Reference Number	Memory Location of Value	Mode of Reference
1	*a*	read
2	*b*	read
3	*c*	write
4	*c*	read
5	*b*	read
6	*d*	write

3.4.3 Cache State Construction

A **cache state** is defined as a possible configuration of cache lines in the cache. In the MAST model, cache state $v_{i,j}$ in the graph represents the j^{th} possible cache configuration immediately after making the reference r_i.

Given the initial cache contents and a reference sequence, there always exists a sequence of cache control instructions (e.g. fetching, replacement, etc.) such that cache state $v_{i,j}$ can be reached by executing reference r_1 to r_i. In addition, any cache line in a cache stage is *live* — any cache line that is dead after reference $v_{i,j}$ is made can be considered as empty or invalid.

Continuing the example in the previous phase (in Figure 3.2), if each value (*a, b, c, d*) is mapped to a different line in cache and the cache size is two, the cache may have one of the following possible cache states in Table 3.2 (to simplify the example, the name of each value is used as its mapped cache line number):

Table 3.2: Possible Cache Configurations for K = 2 and $|M| = 4$

State Number	Cache Configuration	Cache Entries Used
1	$\varnothing$	0/2 (empty)
2	$\{a\}$	1/2
3	$\{b\}$	1/2
4	$\{c\}$	1/2
5	$\{d\}$	1/2
6	$\{a,b\}$	2/2 (full)
7	$\{a,c\}$	2/2 (full)
8	$\{a,d\}$	2/2 (full)
9	$\{b,c\}$	2/2 (full)
10	$\{b,d\}$	2/2 (full)
11	$\{c,d\}$	2/2 (full)

Given a cache of size K and a program of size $|M|$, the maximum possible number of cache states after any reference r_i is:

$$\sum_{j=0}^{K} \frac{|M|!}{j!(|M|-j)!}$$

This number becomes very large for real programs and typical cache size.

However, not all cache states listed in Table 3.2 can possibly exist simultaneously after each reference. After reference r_i, only those cache states that contain lines that are all live after reference r_i are feasible. This is only a small subset of all "possible cache states". Given that m_i is the number of live values at stage i in a particular reference program, the number of feasible cache states at stage i is:

$$\sum_{j=0}^{K} \frac{m_i!}{j!(m_i-j)!}$$

The average number of values that are live simultaneously at a particular point in a typical program is small [AhS86] and is relatively independent of the program size. Since the average m_i is roughly constant across most

reference programs and is relatively small, the complexity can be managed.

Using the same example shown in Figure 3.2 and assuming that no values are live before and after the execution of the given program segment, the live cache lines and feasible cache states after each reference is shown in Table 3.3.

Table 3.3: Live Cache lines And Feasible Cache States

Value Referenced	Live Cache Lines	Feasible Cache States
a	a	$\emptyset,\{a\}$
b	a,b	$\emptyset,\{a\},\{b\},\{a,b\}$
c	b,c	$\emptyset,\{b\},\{c\},\{b,c\}$
c	b,c	$\emptyset,\{b\},\{c\},\{b,c\}$
b	b,c	$\emptyset,\{b\},\{c\},\{b,c\}$
d	d	$\emptyset,\{d\}$

For example, suppose a sequence references 1000 lines. At a particular stage in the reference program, there are only 10 live values. If there are 4 free cache lines and all 1000 lines were assumed to be live, then the maximum number of possible cache states at that stage is 41583792251. In contrast to this extremely discouraging number, if we know that only 16 lines are live, the number of feasible cache states at that stage is known to be at most 2517, which is much smaller than its upper bound limit.

Furthermore, if a direct-mapped cache is used, the average relevant number of live lines is only 4 per cache set and the set size 1, hence, there are *only 5 feasible cache states* for each of the 4 subproblems. In fact, for direct-mapped cache, the average number of feasible cache states for each set is only $m_i/K + 1$ — a *very* small number.

3.4.4 Cache Stage Construction

Cache stage i is the collection of all feasible cache states after memory reference r_i is made. Let $v_{i,j}$ be the cache state j at stage i with the subset of cache lines, S_j, contained in cache at the time immediately after the reference r_i. The cache states that make up stage i, $1 \leq i \leq n$ represent all feasible subsets of program lines contained in cache after memory reference r_i. These vertices can be found in the following manner.

The graph representing the reference string is partitioned into $n+2$ stages, corresponding to the initial stage, r_1, ..., r_n reference stages and the final stage. (The initial and final stages are defined to simplify the graph analysis.)

At stage 0, there is only one cache state, since the initial cache contents is presumably either known or the cache holds no pertinent entries (i.e. $S_0 = \varnothing$; the cache contains no valid entries).

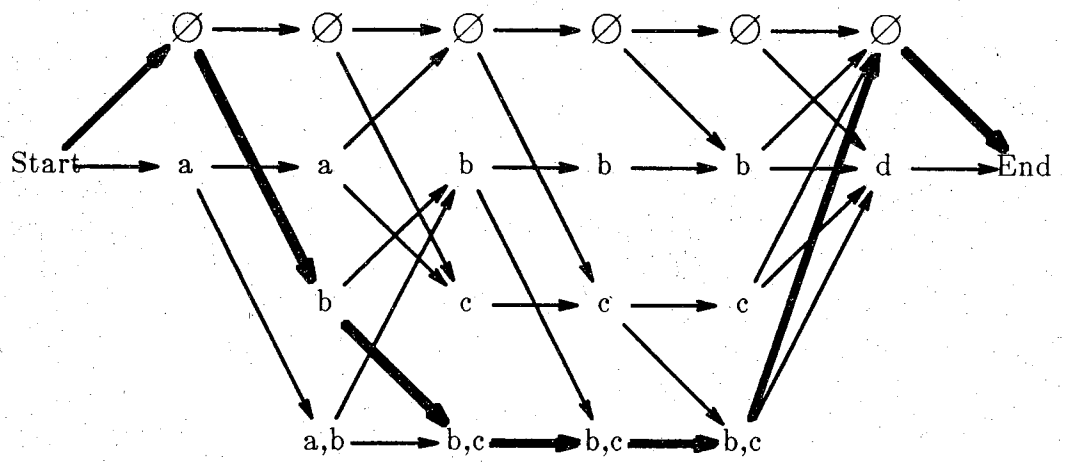
For stage i, $1 \leq i \leq n$, each stage consists of all feasible (reachable) cache states defined by the cache size and the distinct live lines of the memory reference string.

The main reason for constructing these cache stages is that by including all feasible cache states in each cache stage, we can guarantee that there must be one cache state in each cache stage i which shows the cache content after the memory reference r_i. Since all feasible sequences of cache state transitions are included, the optimal cache policy is one which traverses the shortest (cheapest) path from the initial state to the final state. For each memory reference r_i, exactly one cache state j in cache stage i along this optimal path is used.

At stage $n+1$, there is a cache state indicating the final cache contents (when the memory reference string has completed). The cache content S_{n+1} at this final stage is unimportant because after a given memory reference string has completed, there is no reason to prefer one cache state to another — all cache states have the same effect, hence they can be collapsed into a single cache state.

Continuing with the example in Figure 3.2, the final MAST graph is shown in Figure 3.3; each node in the graph represents one feasible cache state and inside the node is the cache content S_j. Note that the number of feasible states in different stages are different. Stage 0 and 7 have one state each, indicating the initial and final stage of the analysis; stage 1 and 6 have two states each; stage 2 to 5 have four states each. Comparing Figure

3.3 with Figure 3.4, which is the MAST graph without the live range analysis, it is clear that live range analysis can dramatically reduce the complexity of the MAST graph.



Stage 0 Stage 1 Stage 2 Stage 3 Stage 4 Stage 5 Stage 6 Stage 7

Ref. a_r Ref. b_r Ref. c_w Ref. c_r Ref. b_r Ref. d_w

Figure 3.3. MAST Graph Pruned by Live-Value Analysis

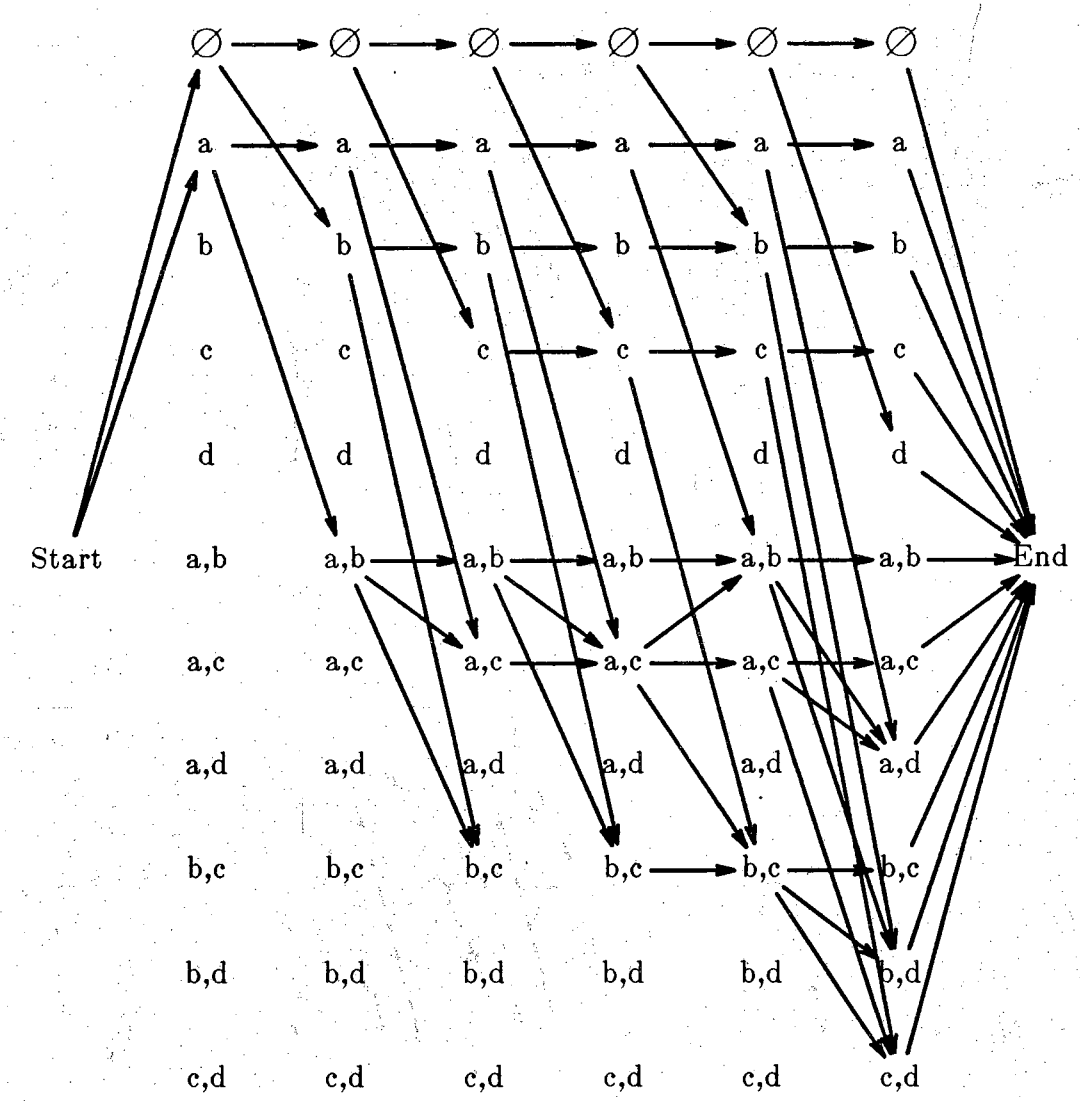


Figure 3.4: MAST Graph with no Pruning

3.4.5 Cache Arc Construction

In the MAST model, construction of an arc from cache state $v_{i,j}$ to cache state $v_{i+1,j'}$ represents a feasible cache state transition in referencing r_{i+1}. The arc leaves a cache state in stage i and points into some cache state in stage $i+1$ which differs only in that reference r_{i+1} may have altered the cache contents. Aside from perhaps having placed a new line in cache, the new cache state may differ in that some line(s) may no longer be live after the reference. In this way, arcs are only created between states in successive stages (and only states which are reachable from the start node need to be considered).

In this phase, arcs for all feasible cache state transitions must be constructed. First, the liveness of lines in any cache state $r_{i,j}$ of stage j are determined. If a line is dead, it is marked as invalid or empty and is treated exactly the same as empty line.

Then, from each cache state $v_{i,j}$, there are three classes of arcs leaving $v_{i,j}$:

No Placement/Replacement:

From each cache state $v_{i,j}$, an arc is constructed to $v_{i+1,j'}$ where $S_j = S_{j'}$. This kind of cache state transition can occur in two situations. First, memory reference r_{i+1} is in the cache and is still live after the reference is made. In this case, the memory reference is directly made from the cache. Second, memory reference r_{i+1} is not in the cache. In this case, the memory reference is directly made from main memory. In both cases, there is *no* change in the cache content.

Placement Without Update:

If $|S_j| < K$, an arc is created to $v_{i+1,j'}$ where $S_{j'} = S_j + r_{i+1}$. This represents a reference which is not available from the cache, but which may be placed in the cache in any entry which was previously not valid (empty). Under typical cache hardware constraints, the cache content will be changed by just one entry. Since there is usually no reason to differentiate between multiple empty entries, only one arc of this type will be drawn from that cache state.

Replacement with Memory Update:

If $|S_j| = K$ then for each $\alpha \in S_j$, let $S_{j'} = S_j - \alpha + r_{i+1}$ and arcs are constructed to $v_{i+1,j'}$. This represents the situation where the cache is full and the next reference is placed into the cache, thereby replacing an existing entry. Since each line in the cache can be replaced, arcs

corresponding to the replacement of each line in the cache are drawn. Any line could be replaced, hence there will be one arc of this type drawn to each cache state where the referenced line is in the cache and the cache is full.

Continuing the example in the previous phase, the graph resulting from cache arc construction is shown in Figure 3.3. Note that a is dead after stage 2. Hence cache stage $v_{2,0}$ and $v_{2,1}$ are actually the same cache state after reference r_2 is made.

3.4.6 Cache Arc Cost Assignment

In the MAST model, the cost associated with an arc represents the expected "relative" cost of going from one cache state to the next in the graph. This cost may be a constant or a variable (as in the case of multiprocessing). Generally speaking, its value depends on the change of the cache content and the delay in accessing the source memory module for the reference r_i.

For each arc in the graph, the cost of the arc connecting cache state $v_{i,j}$ and $v_{i+1,j'}$ is computed and assigned according to Table 3.4. Note that the values of the read and write cost of references are defined by the architecture.

Table 3.4: Reference Types and Costs

Reference Type	Read Cost	Write Cost
Cache	T_{rc}	T_{wc}
Main Memory	T_{rm}	T_{wm}
Cache, Replacing a Live Non-Dirty or Dead Value	$T_l + T_{rr}$	T_{wr}
Cache, Replacing a Live Dirty Value	$T_s + T_l + T_{rr}$	$T_s + T_{wr}$

Each cache state in stage n has one exit arc which enters the final cache state at stage $n+1$. The costs associated with these arcs are 0 (since they do not represent a physical action) or they might be the costs to update dirty entries in cache. In the latter case, the cost of these arcs is a function of the number of dirty lines left in cache.

3.5 Algorithms for MAST Model

The directed graph obtained in previous sections and in Figure 3.3 includes all possible paths corresponding to all feasible (distinguishable) cache control management for the program segment given in Figure 3.2. Each of these paths represents a complete sequence of cache state transitions as the given memory string is referenced.

Therefore, the problem is now the selection of the "shortest path" (least costly allocation sequence) from the initial cache state at stage 0 to the final cache state at stage $n+1$ in the graph. Standard algorithms to find the shortest path problem [AhH74] [Wag76] [Joh77] can be applied to the MAST graph. A more computationally-desirable approach would be to use a pruned search which truncated the search in such a way as to avoid generating most states within the later stages. This will be discussed in Section 4.8.

The optimal path for the previous example is traced in bold lines in Figure 3.3. This optimal path represents the exact cache state transitions used to obtain the lowest cost of referencing the program segment shown in Figure 3.2. Detailed information about type of placement/replacement, place of reference, line number to be fetched, and line number to be replaced are collected and shown in Table 3.5. This information, especially the cache operation for each reference, and the cache line to be operated on, can be embedded into the code generated by the compiler. In this way, optimal cache performance is assured, since the shortest path is, by definition, the optimal set of cache operations.

Table 3.5: Optimal Cache Control Sequence

Ref. String	Bypass/Placement/ Replacement	Place of Reference	Line No. Fetched	Line No. Replaced
Start	$\varnothing$	-	-	-
a_r	Bypass	Main Memory	-	-
b_r	Placement	Cache	b	-
c_w	Placement	Cache	c	-
c_r	Bypass	Cache	-	-
b_r	Bypass	Cache	-	-
d_w	Bypass	Main Memory	-	-

In the MAST graph, the number of feasible cache states is:

$$\sum_{i=1}^{n}\sum_{j=0}^{K}\frac{m_i!}{j!(m_i-j)!}$$

and the number of possible edges is:

$$(K+1)*\sum_{i=1}^{n}\sum_{j=0}^{K}\frac{m_i!}{j!(m_i-j)!}$$

where m_i is the number of distinct live lines at stage i. Since m_i is almost constant, there are $O(n)$ cache states and $O(n)$ edges in a MAST graph for a given cache. Dijkstra's shortest path algorithm for a directed acyclic graph from a single source has an execution time of $O(n^2)$, where n is the number of vertices in the search graph. Hence, the computational complexity of the MAST model is $O(n^2)$. This computational complexity is low compared to many algorithms used in program compilation [AhS86].

3.6 MAST Model in Non-Fully Associative Cache Organizations

In cache design, one of the main factors in obtaining high efficiency is the cache organization. Basically, there are three common types of cache organizations: direct mapping, set associative and fully associative. (Direct mapping and fully associative are actually special cases of set associative where set size is 1 and K, respectively.)

The cache organization that the MAST model assumes so far is fully associative. In this section, applications of the MAST model to direct mapping and set associative cache are discussed. As will be seen in the next three sub-sections, the MAST model discussed so far can be applied to any of these three cache organizations with only minor modifications. In fact, direct-mapped and set associative cache only simplify the computational complexity of the MAST model.

3.6.1 MAST Model in Direct Mapped Cache

This is the simplest, and therefore most commonly implemented, of all possible cache organizations. In this **direct mapped cache**, line i in the memory maps into the line i modulo K of the cache, where K is the size of the cache [HwB84]. Every K^{th} line in main memory is mapped to the same cache line. The direct mapped cache organization is shown in Figure 3.5.

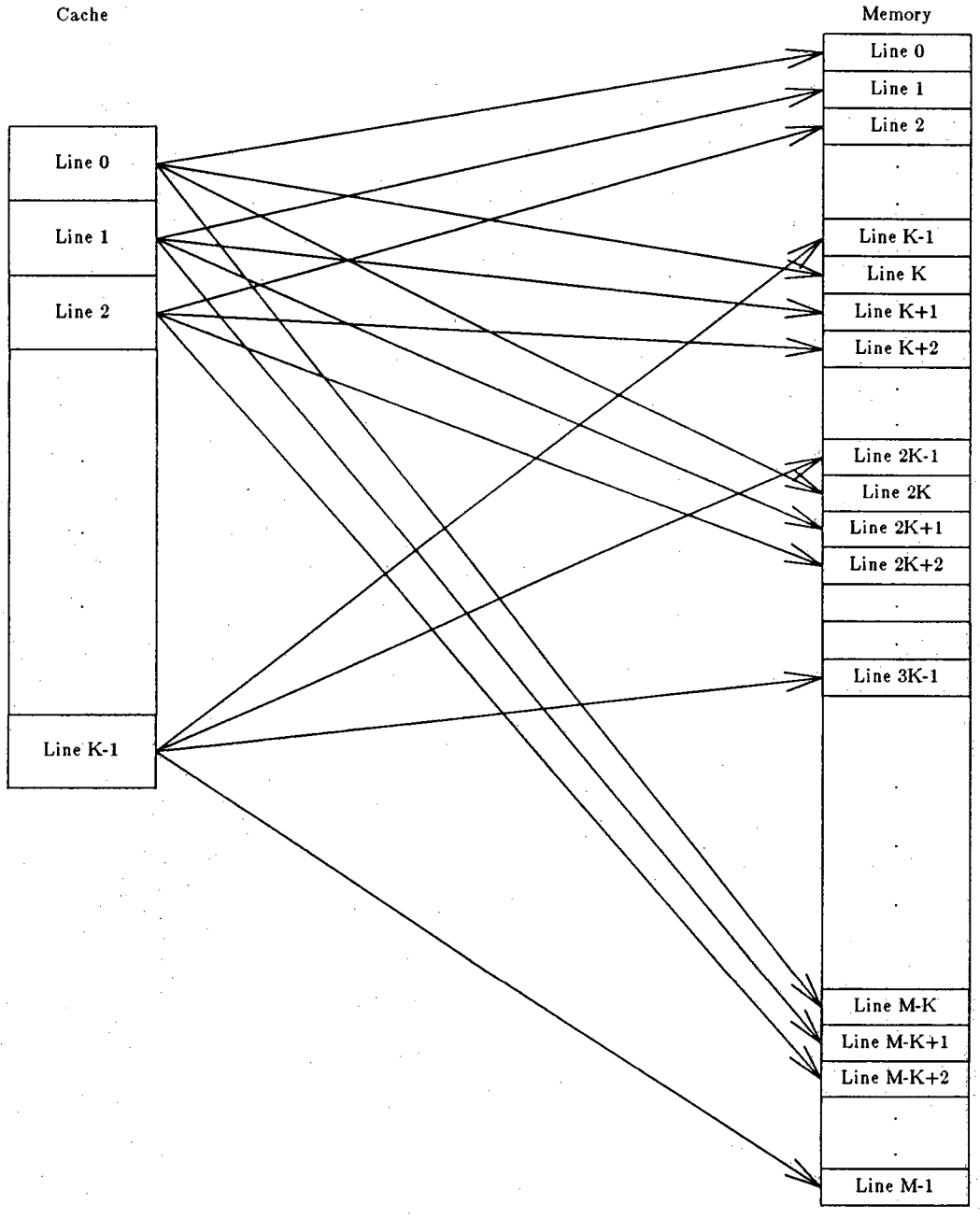


Figure 3.5: Direct Mapped Cache Organization

In the MAST model, direct mapped cache organization can be visualized *as K independent sub-organizations* as shown in Figure 3.6. Each of these K sub-organizations consists of a cache of size one and a main memory of average size MM/K lines (MM is the total size of the main memory). For a given memory reference string with size n, it is also subdivided into K sub-strings, each with average size n/K. In each sub-string, all the line numbers are mapped to the same line in cache — thus reducing the alternatives for performing a reference to just two: either reference directly from main memory or reference using the line in cache. Since each sub-organization and its corresponding sub-string is independent of the others, it can be analyzed separately using the technique described in this chapter.

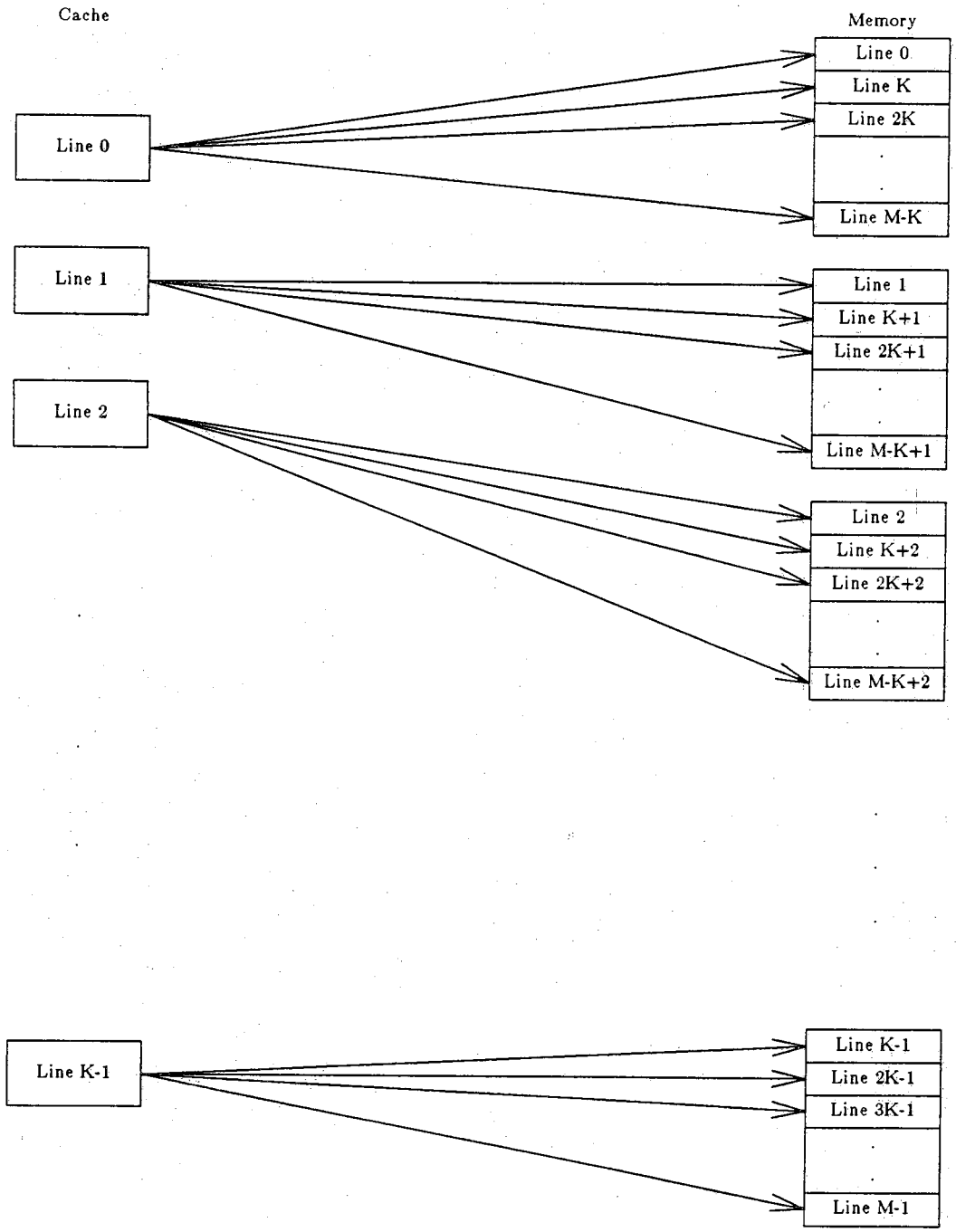


Figure 3.6: MAST Model of Direct Mapped Cache

Using the previous example (shown in Figure 3.2 and Figure 3.3) with line a and c in main memory mapped to the same line in cache and line b and d in main memory to another line in cache, sub-string $a_r c_w c_r$ and $b_r b_r d_w$ are formed. In this case, 2 MAST subgraphs are formed. Each of these subgraphs is analyzed separately and the two independent optimal paths found by the MAST model together provide all information needed to control the cache.

The worst-case complexity of the MAST model in direct mapped cache organization is $O((n/K)^2)$, where n is the length of the reference sequence. This further reduces the computational complexity of the MAST model and makes the model even more practical for very large caches. In fact, *the analysis becomes easier as the cache becomes larger.*

3.6.2 MAST Model in Set Associative Cache

The **set-associative cache organization** with associativity E divides the cache into $S = K/E$ sets, each with E lines per set, where K is the total number of lines in the cache. For reasons of hardware complexity and performance tradeoff, E is often restricted to a small number, typically one or two, and seldom greater than four [Smi78c]. A line i in main memory can be cached in any line belonging to the set i modulo S [HwB84]. Set associative cache organization with set size of two is shown in Figure 3.7.

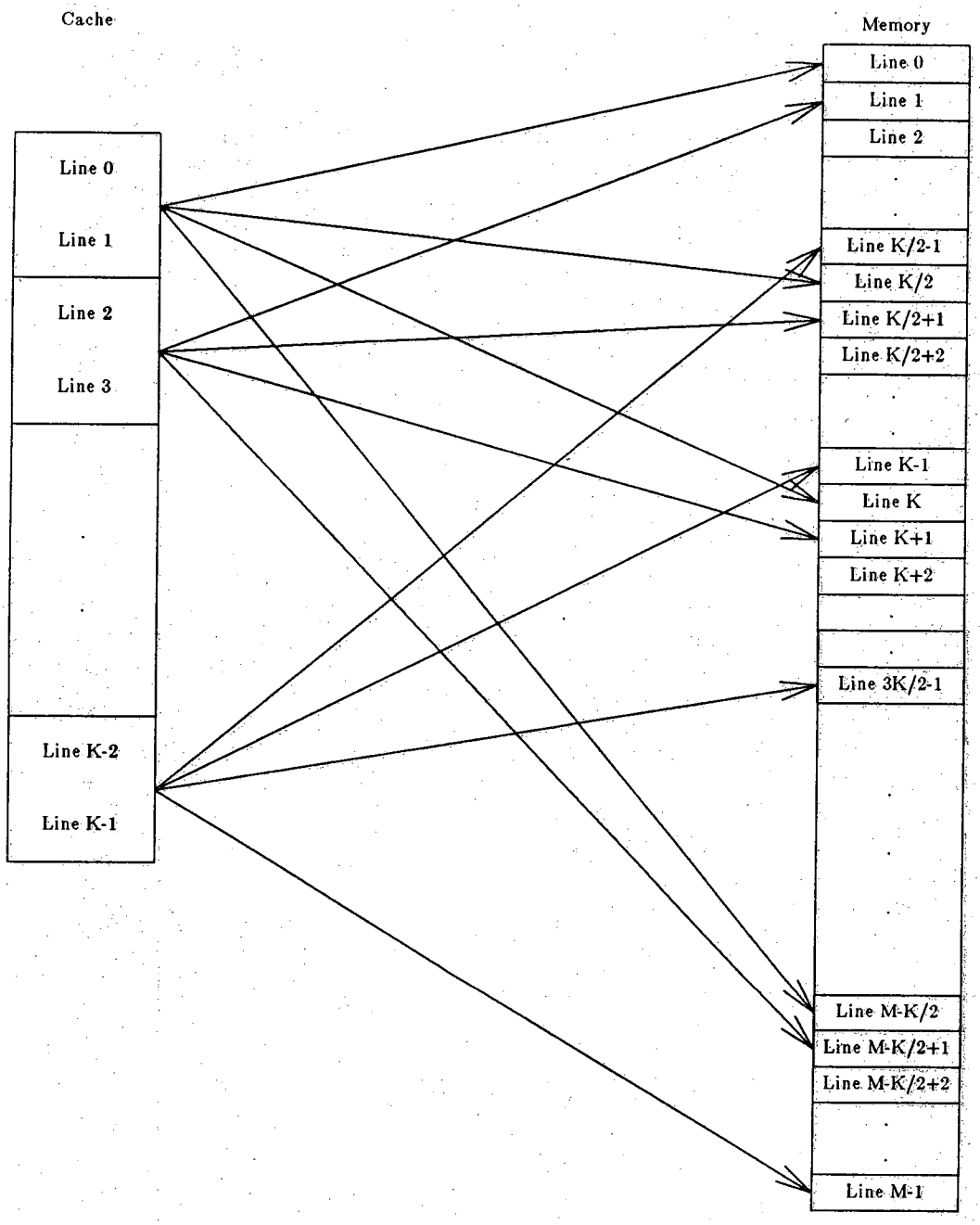


Figure 3.7: Set Associative Cache Organization

In the MAST model, set associative cache organization can be visualized as K/E independent sub-organizations. Each of these K/E sub-organizations consists of a cache of size E and a main memory of size MM/S (MM is the total main memory size). The memory reference string with size n is also subdivided into S sub-strings, each of which has all its symbols mapped to the same cache set and has an average size n/S. Since each sub-organization and its corresponding sub-string is independent of the others, it can be analyzed separately using the technique described in the previous sections. An example of set size of 2 is shown in Figure 3.8.

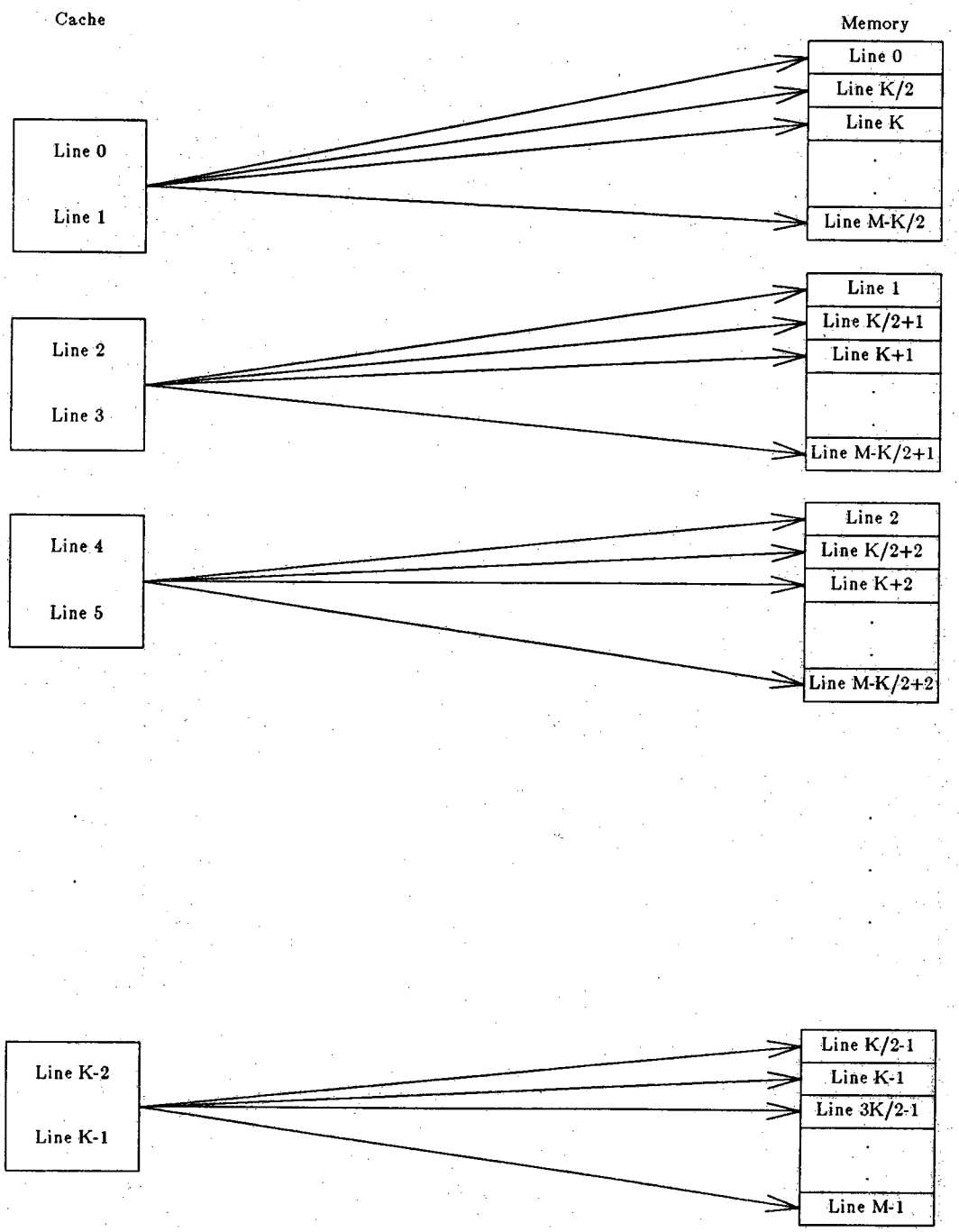


Figure 3.8: MAST Model of Set Associative (Size Two)

The worst-case complexity of the MAST model in the set associative cache organization is $O((nE/K)^2)$, where n is the length of the reference sequence. Since E is a constant in a given cache organization, the MAST model is still practical. For example, if E is 2, then the worst-case complexity is just $O((2n/K)^2)$. Again, it is useful to note that, on the average, increasing the cache size but not the set size will simplify the problem.

3.6.3 MAST Model in Fully Associative Cache

Fully associative cache organization permits any line in main memory to be mapped into any line in cache. In other words, set size E is equal to the cache size K. Fully associative cache organization is shown in Figure 3.9.

The techniques described in this chapter can be used to analyze the fully associative cache organization. Fully associative cache organization in the MAST model is shown in Figure 3.10. The worst-case complexity of the MAST Model in a fully associative cache organization is $O(n^2)$.

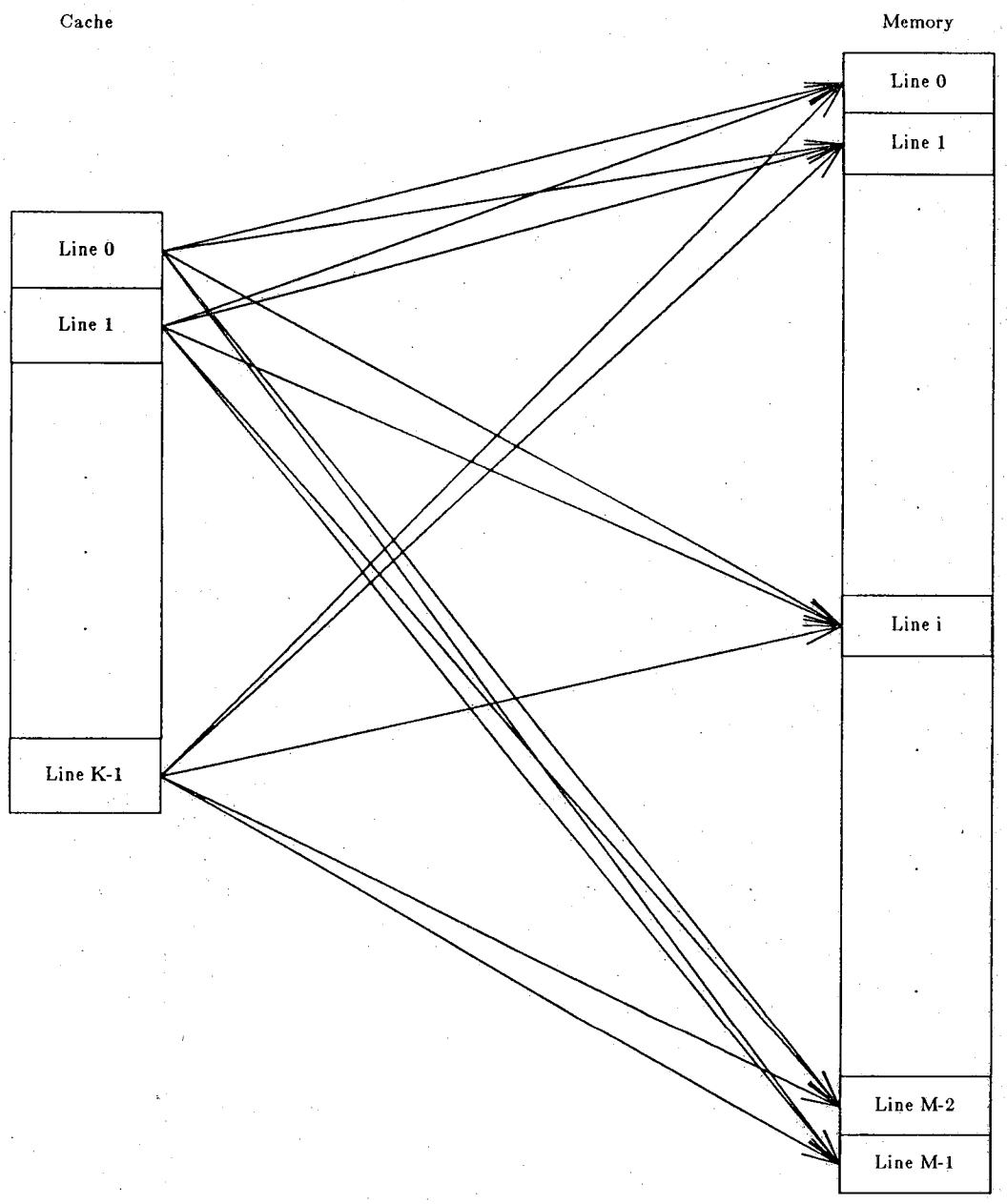


Figure 3.9: Fully Associative Cache Organization

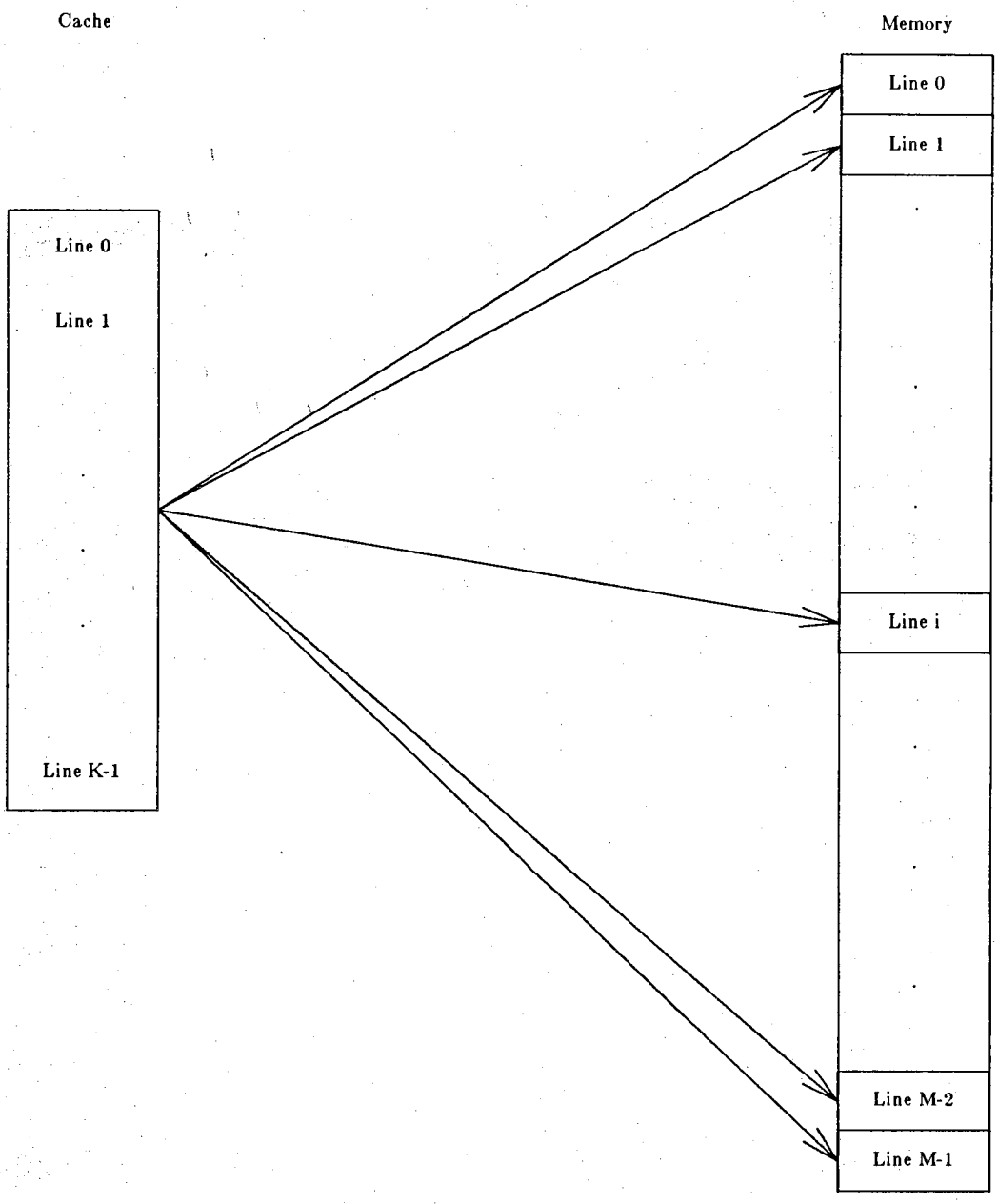


Figure 3.10: MAST Model of Fully Associative Cache

3.7 Conclusion

In this chapter, a compiler-driven cache management scheme, called the MAST model, is proposed to optimize system performance based on execution time. Since this model is based on machine level state transitions and the sequence of program memory references is considered in the model, cache can be managed more "intelligently A large potential improvement of cache performance can be obtained.

With the graph formulation of a program and shortest-path based analysis, this model *guarantees optimal cache performance.* Given a reference string which is known at compile-time, cache control are optimally managed. Although this precise knowledge of the reference string is unlikely to be available at compile-time, there are techniques developed to handleg various program structures. This is the main focus of Chapter 4 — the generative uses of the MAST model in different program structures.

The use of the MAST model in different cache organizations is also discussed. Most computer systems employ very simple cache with small set size and small line size. For these simple cache organizations, the computational complexity of the MAST model is very low. Since caching is based on lines, each of which may contain a number of memory locations, the number of distinct lines in a reference program is reduced and the computational complexity is further reduced. The fact that the static code of a program is used in the analysis also makes the MAST model computationally feasible because dynamic code size is always larger than static code by a factor of 1000 or more. Moreover, heuristic algorithms can always help us to reduce the graph analysis time, thereby improving compile time. Alpha-beta pruning can dramatically reduce the graph analysis time without sacrificing optimality. Various pruning techniques for the MAST model will be discussed in details in Chapter 4.

CHAPTER IV
GENERATIVE USES OF THE MAST MODEL IN PROGRAMS

4.1 Introduction

In the previous chapter, a compiler-driven cache management model based on machine level state transitions (MAST) is proposed. Using this new model, it is possible to determine the optimal cache performance, and the corresponding cache control operations, for a reference program consisting of a compile-time known reference sequence. This type of reference program is effectively the reference sequence found within a basic block or from a program execution trace. In this chapter, we discuss how the MAST model can be extended to general program structures such as loops and branches and to generate optimal (or nearly optimal) *static* cache control for *runtime* cache management.

To extend the MAST model so that it may be used to *generate*, rather than merely to predict, optimal cache performance, it is necessary that the model deals not only with arbitrary program structures, but also it should only require *information that can be obtained by a compiler.* Such information is, by definition, imperfect. For example, the concept of references being **unambiguous** — known to refer to either same object or disjoint objects — is an unacceptable simplification. Consider a program fragment which refers to *a[i]* and *a[j]*, where the values of i and j are read as input at runtime. It is generally impossible for the compiler to know whether these two names refer to the same or to different objects. The causes of this compile-time ambiguity include:

- The exact sequence of memory references at execution time may not be known. This variation is caused by runtime branching decisions.
- The actual physical memory addresses, and values stored in them, may not be certain until runtime. This is due to dynamic memory allocation and mapping as well as the possible use of data structures whose internal arrangement depends on computed values (such as hash

tables).

Rather than requiring the compiler to provide perfect, unambiguous, information, a feasible generative application of the MAST model must work with these fuzzy constraints. Of course, under such circumstances, the MAST model can only insure optimality in terms of the constraints it is given — less precise or incorrect constraints will result in suboptimal performance at runtime.

To obtain a reasonable complete and accurate constraint information at compile time, we extend the MAST model to handle the following structures as special cases:

- Loops — How many times does a loop iterate?
- Branches[1] — Given branches diverging to various different segments of the program, which way does the branch actually go? Eventually, all these diverging control flow paths converge (at least at the program's termination) — where and how?
- Subroutines and functions — Since there might be loops, branches, and nested calls in a subroutine or function, what is the reference sequence for a subroutine? Notice that subroutines and functions are essentially equivalent in terms of this complication, although recursion is a major additional complication in either case.
- **Ambiguous references** — Programs refer to values using names, but at execution time all references are made in terms of addresses. At compile-time, the mapping of names to address may be ambiguous. This ambiguity comes in two sources. First, it might not be possible to predict where a name will reference. Second, it might not be possible to determine when two names may actually map to the same address. Note that only the second kind of ambiguity has been widely acknowledged — recognition of the first is a contribution of this thesis.

The main focus of this chapter is to extend the basic MAST model described in Chapter 3 to programs with these structures. In Section 4.2, an

[1] Forward branches, henceforth simply "branches," are associated with conditional execution, whereas backward branches are used to return to the start of the next iteration of a loop. It is assumed that conceptually unnecessary branches, as are associated with "spaghetti code," will be eliminated by code straightening [AhU77] [AhS86] [Die87] prior to MAST analysis. Hence, for the MAST model, each branch is either a loop "back edge" or a forward branch.

important concept of the MAST model, called the **MAST cache cut point**, is introduced. Extensions of the MAST model in this chapter are all based on this concept. Extension of the MAST model to loop structures is discussed in Section 4.3. In Section 4.4, uses of the MAST model in the presence of divergence-of-flow and convergence-of-flow are given. The handling of subroutine calls in the MAST model is presented in Section 4.5. Section 4.6 discusses the handling of ambiguous references in the MAST model. In this Section 4.7, a new memory structure, called the **CReg** (Cache-Register), is proposed as a hardware solution to the ambiguous alias reference problem. Various pruning techniques to reduce the computational complexity of the MAST model are given in Section 4.8. Section 4.9 discusses the implementation of the MAST model. Finally, the chapter summarizes in Section 4.10.

Before the discussion of extensions of the MAST model to handle these program structures, it is necessary to first introduce the concept of a **MAST cache cut point**, which serves as a tool to divide a reference program into several non-overlapping smaller reference programs.

4.2 MAST Cache Cut Point

A **MAST cache cut point** in a reference program is defined as a stage in a reference program where the cache state at that stage in the MAST model allocation can uniquely be determined by studying at most a portion of the program around that stage — without the complete formulation or analysis of the MAST model. A MAST cache cut point is said to be **optimal** if the cache content at the MAST cache cut point is on the optimal allocation sequence. An **artificial MAST cache cut point** is a MAST cache cut point where the cache content is explicitly assigned to a specific cache configuration at compile-time and this configuration might or might not be on the optimal allocation sequence. In the rest of the discussion, the words "MAST cache" in the term "MAST cache cut point" might sometimes be omitted for simplicity.

A MAST cache cut point divides a reference program into two independent smaller subprograms, each with a boundary cache configuration, one at the start and the other at the end. References which are on different "sides" of a cut point have no effect on each others cache allocation.

Examples of situations in a reference program which cause cache stage i to be a MAST cache cut point are:

- When there is no live value at cache stage i. In this case, the cache content must be empty. Any stage of a reference program immediately following a subroutine call might belong to this class[2]. (This cut point is optimal if the subroutine is large enough so that any useful lines in cache before the call are already flushed from the cache when the subroutine returns to the caller. Otherwise, it can only be considered as an artificial cut point.)
- When every live line at cache stage i is known to be either in cache or in main memory, and the number of lines in cache is less than the number of cache lines available. For example, in traditional cache design, any datum must be put in cache before it is referenced. Hence, if the number of live lines at cache stage i is not greater than the number of cache lines available, the optimal cache allocation is to put these live lines in cache.
- When a region of a reference program constrains the cache contents at the entry and exit of the region. The constraint may be, for example, that the region is the body of a loop which iterates many times, hence the entry cache contents and exit cache contents should be the optimal contents found by just considering cache allocation within the loop body region (ignoring the surrounding code).

These situations occur very frequently in programs. Previous research on program behavior showed that about 40 percent of instruction executed in a program is found in loops and about 13 percent of the statically counted HLL statements *call* statements [AlW75] [Kat83].

This MAST cache cut point is extremely important concept in the extensions of the MAST model to programs with branches, loops, and subprograms. With this concept, the MAST model analysis of a reference program is *equivalent* to the independent MAST model analysis of each of the regions in the reference program bounded by valid MAST cache cut

[2] This assumes that the cache architecture does not save the cache contents before a subroutine call and restore them on return. If saving and restoration of cache contents before and after a subroutine call is used, the subroutine might be considered as "transparent" to the caller routine. In other word, there is no effect on the callee routine due to the caller routine.

points. Since each region is typically a basic block (a sequence of codes with only a single entry and exit point), the MAST model presented in the last chapter can be applied.

Further, the types of MAST cache cut points mentioned above occur quite often in reference programs. Since the MAST model analysis complexity generally grows faster than linearly with the length of the reference program, breaking reference programs into independent regions bounded cut points can significantly speed up the MAST model analysis.

Both artificial and optimal MAST cache cut points serve the same function — to simplify the cache allocation problem. Since artificial cut points can be arbitrarily introduced, they can greatly simplify the cache allocation problem. The tradeoff is the sacrifice of the optimality of the solution. However, the cache performance loss is only very small — the worst case is a cache cold start for each artificial MAST cache cut point.

In the next sub-section, rules to determine or find MAST cache cut points are discussed.

4.2.1 Determination of MAST Cache Cut Points

The presence of a MAST cache cut point at cache stage i in a reference program implies that each live line at stage i is known to be either in cache or in main memory. Hence, there are two questions that need to be answered before the determination of a MAST cache cut point. For each live line λ at cache stage i,

- what are the relative benefits and overhead which line λ brings if it is put in cache?
- if the cache space is not large enough to hold all these "worthy" lines, what are their cache placement priorities?

Solutions to the above questions at every cache stage in a reference program can be given by the MAST model. However, there are situations where they can be answered without going through the model. To explore this, we first need several definitions.

Definition 4.1. Cache Viability

A line λ is a **viable** candidate for being placed in cache *iff* the minimum cost for placing λ in cache and referencing it from there κ times is less than the maximum cost for making κ references directly

from main memory, for some κ which is greater or equal to the maximum number of references made to line λ in a reference program.

Definition 4.2. Cache Strong Viability

A line λ is **strongly viable** over a reference program segment between stage α and stage β *iff* the maximum cost for referencing λ through cache (assuming a line is available for) for all references between stages α and β is less than the minimum cost for referencing λ in any other way over that same period (such as bypassing on some or all references).

Hence, the viability of a line λ indicates whether loading line λ in cache before referencing it improve s system performance. With these definitions, the following two rules need to be satisfied before any line is placed in cache:

Rule 4.1. Cache Inviability

A line λ which is not viable should never be placed in cache in the optimal cache allocation.

Rule 4.2. Cache Allocation Requirement

Each strongly viable line λ in the range between cache stage α and cache stage β, will be placed in cache on all optimal paths between cache stages α and β if the number of strongly viable lines over the range from cache stage α to cache stage β is less than the cache size.

The criteria used in these two rules for cache line placement is quite different from the criteria used in current cache design, in which *every* referenced line is placed in cache before it is referenced.

Two more rules about the use of the MAST cache cut point are:

Rule 4.3. MAST Cache Cut Point

Given a particular reference program, if it is known that cache stage α is a MAST cache cut point, then the optimal cache allocation from stage 0 to stage α is independent from the optimal cache allocation from stage α to final stage. The path from stage α to the final stage may be considered a separate allocation problem whose initial state has all viable values in cache.

Rule 4.4. Loop Cut Points

Given a particular reference program which contains a loop with a relatively high expected number of iterations, only lines whose U-D or D-U chains are rooted in the loop body need be considered live within

the loop — in effect the loop body is "cut" from the rest of the program. If the optimal solution for the extracted loop body is unique, the cache allocation problem for the entire reference program is modified by considering the loop reference stages to be replaced by a single stage. The cache content for that stage can only be a superset of the cache contents which are found to be optimal at the extracted loop's edges.

An example of applying these rules to find MAST cache cut point might help clarify this concept. A possible reference sequence is given in Table 4.1. With live range analysis, the number of simultaneous live lines is found. Note that there is no live line in between cache stage 6 and 7, stage 9 and 10, and stage 11 and 12. Hence, these are the three MAST cache cut points and the reference sequence is divided into four subsequences:

- stage 0 to stage 6,
- stage 7 to stage 9,
- stage 10 to stage 11, and
- stage 12,

and each sub-sequence can be analyzed separately, yet the optimality of the cache performance can still be achieved. Furthermore, if data need to be place in cache before it is referenced, stage 1, 6, 7, 8, 9, 10, 11, and 12 are all MAST cache points.

Table 4.1: Line Reference Sequence

Stage No.	0	1	2	3	4	5	6	7	8	9	10	11	12
Value	A	B	A	C	A	C	B	A	A	A	B	B	B
Read/write	r	w	r	w	r	r	r	w	r	r	w	r	w
No. of Live	1	2	2	3	3	2	1	1	1	1	1	1	1

With the MAST cache cut point, the MAST model discussed in the last chapter can be extended to be used in programs with other program structures.

4.3 Loops

An occurrence of a **loop** is defined as a sequence of instruction executions in which a particular sequence of distinct instructions is successively repeated [Kob80] [Kob84]. Distinct instructions in a cycle do not necessarily have contiguous addresses. A loop must be entered in the beginning, but it may be terminated in the middle of the loop (i.e. a loop may have more than one exit point). Loops can be defined recursively, with the innermost loop first. The **nesting level** of a loop is defined as one plus the highest level of the subloops within the loop, or as level one if it does not contain any subloops. Some examples of loop structures are for...loops, while...loops and repeat...until.

It is generally agreed that loops are one of the most important control flow structures in programs. A large percent of program execution time is spent within loops of various kinds. Previous research on program behavior showed that about 40 percent of program execution time is within loops [AlW75] [Kob80] [Kob84] [Kat83].

Cache performance within loops is one of the few key factors which determine the efficiency of a cache design. The principles of temporal and spatial locality of reference suggests that information currently being used (and is in cache) has a high probability of being re-used in the near future. *Iteration of a loop means that a sequence of references is repeated.* Hence, besides the fact that a large percent of program execution time is spent in loops, there is a "good match" between the working principle of cache and that of reference behavior of loops.

However, if cache management in loops is not handled properly, the presence of cache actually degrades system performance instead of improving it [SmG83] [SmG85]. An example might help understand this. Suppose the reference sequence of a loop body is a,b,c,d,e and the cache size is four, set size four and line size one. If the cache uses LRU as its replacement policy, it always replaces the datum which is going to be referenced next after the reference d of first iteration of the loop is made. This is the worst cache performance that any replacement scheme can

possibly have — *even worse than the performance without cache.* This performance loss can easily be regained by placing *abcd* in cache during the first loop iteration, "freezing this cache state", and referencing *a, b, c, d* from cache and *e* directly from main memory (bypassing the cache) once the cache is full.

Large cache performance loss in loops using current cache replacement schemes have been observed by some cache researchers [SmG83] [SmG85] [ChD87]. Attempts have also been made to perform loop transformations in a reference program so that "most" loops can be placed entirely in cache [Abu79] [Tha81] [Mac83] [Mac87]. However, there are always loops with size larger than the cache size and no further subdivision of the loop body is possible — and this is especially true for small caches. Current cache management schemes simply cannot handle loops with size larger than cache size properly. It is surprising to find that even simple "fixes" like freezing and bypassing cache in loops are overlooked by all current cache management schemes.

On the other hand, the MAST model described in Chapter 3 can easily be modified to handle loops with arbitrary loop and cache size. This is the main focus of this section. In Section 4.3.1, the dynamic characteristics of loops are given. Analysis of current cache management in loops is discussed in Section 4.3.2. Extension of the MAST model to loop structures is presented in Section 4.3.3. Three strategies proposed are:

- loop unravelling,
- loop unrolling, and
- loop intact.

Comparisons between cache performance in loops using current cache management schemes and the MAST model is made in Section 4.3.4.

4.3.1 Dynamic Characteristics of Loops

In this section, results from previous research on dynamic memory reference behavior of loops are presented. This presentation is by no means complete — different benchmark programs running on different computer architectures might have different statistical results. Instead, it mainly serves the following purposes:

- To show that a high percentage of program execution time is within loops.
- To highlight key features about a loop's reference behavior.
- To show that the situation where current cache management schemes have the poorest performance — loop size is slightly greater than the cache size — is very common.

Kobayashi studied instruction reference behavior and locality of references of programs and collected detailed statistics about dynamic characteristics of loops [Kob80] [Kob83] [Kob84]. The benchmark programs that he used in this study consisted of various compilers and application programs, including FORTRAN programs with predominantly long and/or short floating point instructions, COBOL programs, FORTRAN and COBOL compilations, and PL/I optimizing compilers. He collected forty-one instruction traces on these compilers and application programs running on the IBM System/370, each trace with length from 0.2 to 6.4 million references.

The memory reference statistics about dynamic characteristics of loops which he collected and its impact on cache design are summarized as follows:

- A high percentage of program execution is found within loops: roughly about 50-60 percent for application programs and 21-36 percent for compilers (other researchers also found similar results [PeS82] [Kat83]). Moreover, most cache misses in LRU (one of the two "best" cache replacement schemes) are within loops [Puz85]. This suggests that a large performance improvement might be obtained if current cache management schemes can be modified to handle loops more intelligently.
- For those references made within loops, about 54 - 68 percent from application programs and 76 - 85 percent from compilers were found within loops of cycle length 128 bytes or shorter. This has two important implications:
  - An instruction cache of size 128 bytes (or slightly larger) might be large enough for most practical purposes. However, in current cache design, an instruction cache of size 1K - 4K bytes is always considered as "too small" to give any performance improvement [Hil83] [HiS84].

- In VLSI processor designs with small on-chip cache, the situation where the loop size is slightly larger than cache size might occur frequently. Hence, if current cache management is used, the worst case performance described in [SmG83] [SmG85] (also discuss in details in the next section of this chapter) results.

- The average number of repetitions of a loop is relatively independent of the cycle length. Within one loop, the average number of iterations ranges from 2.3 to 13.3, but it might go up to about 57.

Although these numbers are program-dependent, they indicate that within a loop, the cache contents are very much determined by the loop body and only slightly influenced by memory references outside the current executing loop.

The above statistics about dynamic characteristics of loops once again confirms the importance of the loop structure to cache management and motivates research for intelligent cache management.

4.3.2 Analysis of Current Cache Management in Loops

Smith and Goodman studied the performance of instruction cache replacement policies and organizations in loop structures [SmG83] [SmG85]. They found that random replacement always performs better than LRU and FIFO in loops, and in certain situations, a direct mapped or set-associative cache may perform better than a full-associative cache organization. In particular, they were interested in the case where the cache size is slightly less than the loop size.

Their results and its implications to cache design are summarized as follows. Given a reference program loop of size λ and a fully associative cache size of κ:

- $\lambda \leq \kappa$

  The steady state cache hit ratio for FIFO, LRU, random and optimum[3] replacement is 1. Since the cache size is larger than the loop size, the entire loop is in cache after first loop iteration. Hence, any reasonable cache management should have the same performance.

[3] The optimum replacement policy here refers to the Belady's MIN algorithm [Bel66]. In essence, it refers to a replacement policy which replaces the line that will be referenced the furthest in future.

- $\lambda > \kappa$
  - Both FIFO and LRU have a steady state line hit ratio of 0. This happens because the cache line replaced is the line which has the smallest reference distance[4]. Hence, cache lines are continually replaced just before they are referenced and the steady state performance is always cache miss.
  - The steady state line hit ratio of Belady's MIN algorithm ("optimal" algorithm) [Bel66] is $(\kappa\text{-}1)/(\lambda\text{-}1)$. This is the best performance that current cache management schemes can ever have. However, there are problems associated with this policy:
    - Current cache hardware does not have capability to realize this policy.
    - This policy is not optimal if total time execution is the measuring parameter (see Section 4.3.4). Furthermore, there exist some simple (both hardware and software) implementable algorithms which have better performance than MIN algorithm (these will be discussed in Chapter 5).
  - The random replacement scheme always performs better than both LRU and FIFO because cache hit ratio of 0 is the worst performance than any algorithm can ever have. Moreover, if $\lambda = \kappa+1$, the random replacement policy gives a steady state cache line hit ratio of $(\kappa\text{-}1)/\kappa$.
- With a direct mapped cache:
  - if $\lambda < \kappa$, the line hit ratio is 1;
  - if $\kappa < \lambda < 2^*\kappa$, the line hit ratio is $((2^*\kappa\text{-}\lambda)/\lambda)$;
  - if $2^*\kappa < \lambda$, the hit ratio is 0.

  This shows that direct mapped cache can sometimes perform better than set-associative cache using LRU or random replacement schemes.

The above derivations are only for simple instruction loops. For nested loops or data reference loops, a closed form for cache performance is difficult to obtain. However, the argument still holds.

---

[4] The **reference distance** of a datum is the minimum number of references made before the datum is referenced again.

4.3.3 MAST Model on Loops

There are at least three approaches that MAST model can use to handle loop structures. They are:

(1) loop unravelling,

(2) loop unrolling, and

(3) loop intact.

The first two approaches require program transformation while the third approach does not change the given program.

4.3.3.1 Loop Unravelling

Loop unravelling is the technique where a loop whose body iterates a small, compile-time constant, number of times, may be unravelled to create a loop-free reference program [DoJ79] [Fis81] [Ell85] [Die87]. In effect, this approach transforms a loop into a large basic block [Hsu87]. Figure 4.1 shows a loop body that is unravelled several times. With this technique, the MAST model described in Chapter 3 can be applied directly to the resulting basic block without any extension.

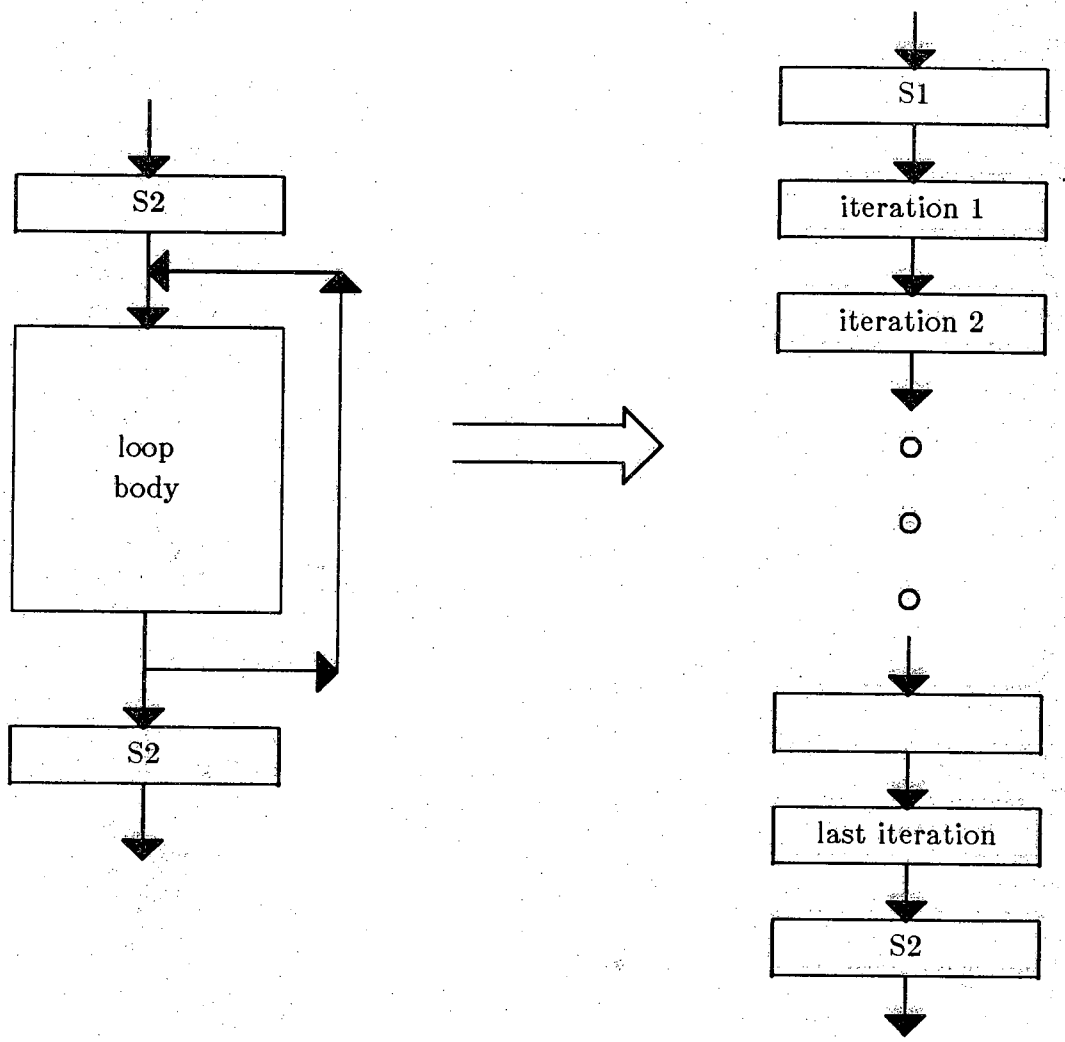


Figure 4.1: Loop Unravelling

There are several advantages to unravelling a loop. Since code representing different iterations of a loop body may incorporate different cache control directives, unravelling a loop increases the flexibility in specification of cache control.

Loop unravelling eliminates branch and index-update instructions that need to be executed. Furthermore, possible instruction stream pipeline flushing due to branches (if the pipeline machine does not handle loop properly) is avoided. Cache prefetching is also more effective in a basic block than in a cache loop with branches.

Once the loop is unravelled, more code optimizations can be performed in the resulting basic block. For example, Figure 4.2 shows a loop before and after loop unravelling. *A[i]* can be considered as a common subexpression; thus it can be allocated in a register and reload of *A[i]* for every *use* of *A[i]* is avoided.

```
for (i = 1; i <= 10; i++) {
    A(i) = A(i-1) + B(i)
}
```

(a) Before Unravelling

```
A(1) = A(0) + B(1)
A(2) = A(1) + B(2)
A(3) = A(2) + B(3)
:
:
:
A(10) = A(9) + B(10)
```

(b) After Unravelling

Figure 4.2: Loop Unravelling and Register Allocation

However, not all loops can be unravelled completely. Generally, the number of iterations that a loop will go through is unknown at compile-time. Further, since loop unravelling increases the code size, it might have some undesirable side-effects. For example, instruction fetch caching may be rendered ineffective because of the reduction in locality. The other side-effect is to increase the amount of static code that the MAST model need to analyze.

4.3.3.2 Loop Unrolling

Loop unrolling is the technique, where a loop is unrolled to separate the first and last iterations from the middle iterations [Die87]. Figure 4.3 shows the unrolling of a loop.

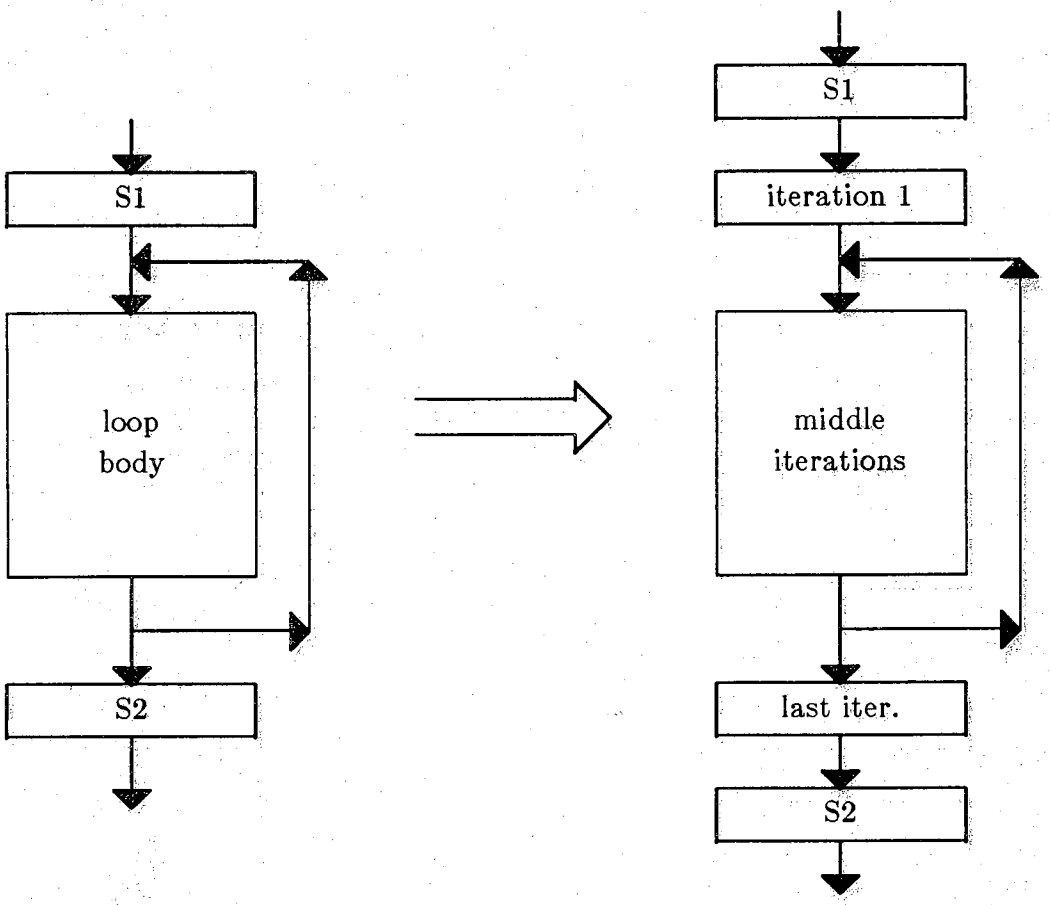


Figure 4.3: Loop Unrolling

When the number of iterations is not a compile-time constant, or the number of iterations is very large, complete loop unravelling is not possible. However, optimal cache control inside a loop body is still closely related to the loop inter-iteration dependence structure.

There are two loop unrolling guidelines to determine the optimal cache control sequence inside a loop:

- The optimal cache control sequence of the loop body at eac middle iteration should be the same. The only exception is the pointer references in data cache. This will be discussed in details in Section 4.6.
- The cache contents in the beginning and at the end of each loop iteration should be the same. This is because when a loop iterates back, cache contents need to match each other.

Hence, the optimal cache contents for each cache stage inside a loop should only be determined by what is referenced inside a loop body, and with the above two guidelines. No references outside a loop body need to be considered.

The only exception, perhaps, is the the first and the last iterations of a loop. The first iteration is special because it places the appropriate items into cache, replacing any assignments made before entering the loop. With this, a smooth transition of cache configurations is obtained when a loop is entered. It is also desirable to isolate the last iteration because the last iteration can make a smooth transition to the cache configuration desired for the code following the loop.

Loops whose iteration deciders [Die87] are loop invariant expressions can easily be unrolled in this way; loops in general may not easily permit the last iteration to be separated from the middle iterations, but, in any case, it is easy to isolate the first iteration.

The advantage in performing this separation is that for each of the middle iterations, it is typically optimal to have identical cache configuration at the same point in each iteration. Hence, the cache allocation for middle iterations of a loop can be found by just considering the loop body itself and this is independent of the number of iterations made by the loop. Since the same MAST cache cut point bound the loop body, a loop in a reference program can be replaced by the MAST cache cut point in the analysis. (i.e. a reference program is divided into three pieces.)

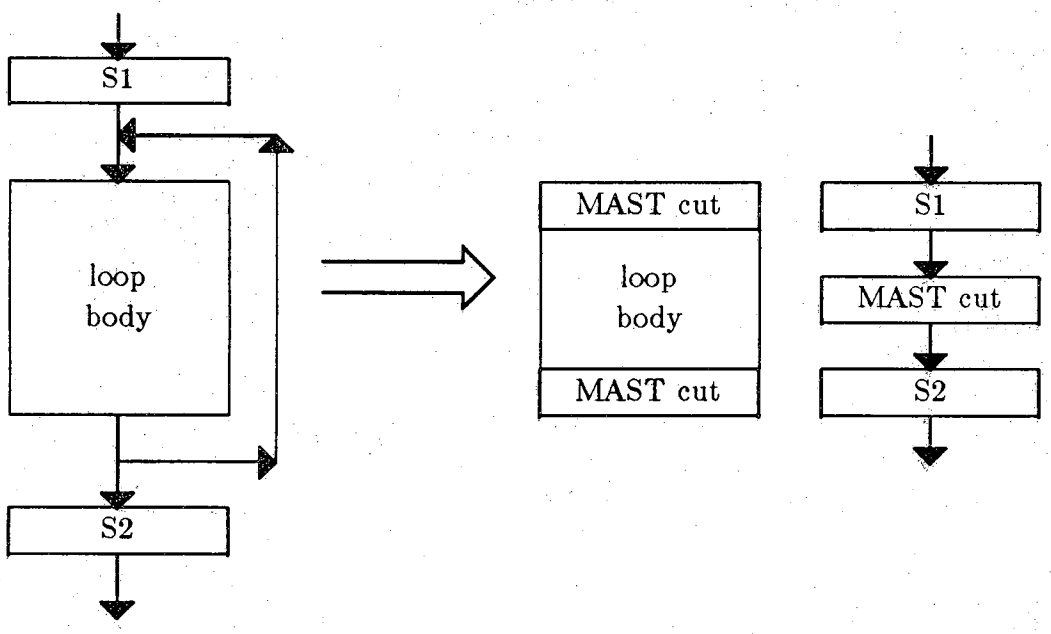


Figure 4.4: Loop Unrolling in MAST Model

Both loop unravelling and loop unrolling require some kind of program transformation. Obviously, the cache performance resulting from unravelled and unrolled version of a loop is not the same.

4.3.3.3 Loop Intact

This is the technique where optimal (or nearly optimal) cache control may be obtained for a loop in its original form — no separation of loop iteration is allowed. In this scheme, one can imagine a burst of stores and loads just prior to entering the loop and a burst of stores and loads just after exiting (see Figure 4.5). This is nearly optimal, assuming the loop uses all cache entries and iterates many times.

Effectively, cache allocation within a loop would be treated almost independently of the allocation for the code that surrounds it. Further, the cache control for each iteration must be the same. Hence, a loop is bounded by the same MAST cache cut point, which can be found independently from the rest of the program. Then, the loop in the reference program can be replaced by this cut point found in the MAST model analysis of the loop body. This greatly simplifies the analysis, since the complexity of the

analysis does not increase in proportion to the number of iterations and the program is sub-divided into three segments.

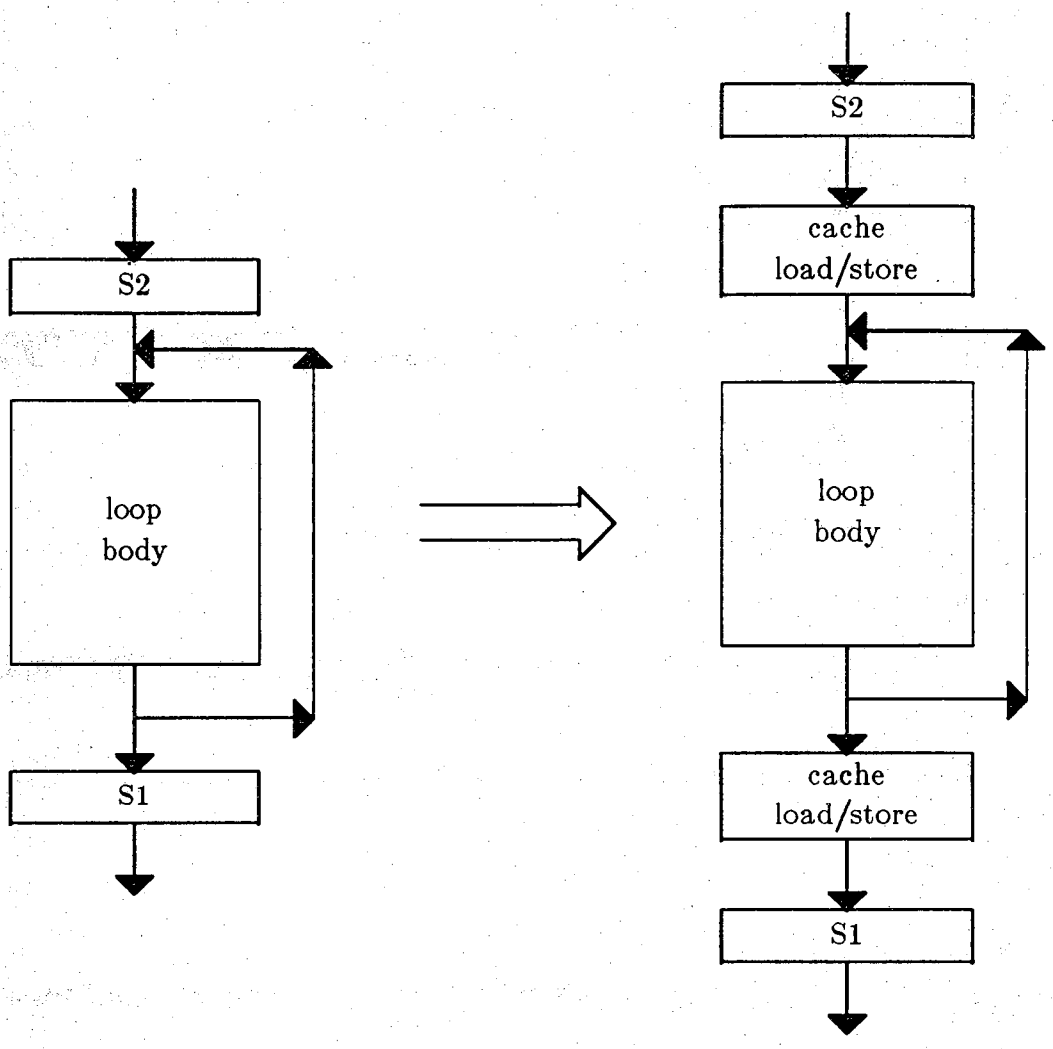


Figure 4.5: Loop Intact

Although bursts of store and load theoretically exist before entering and just exiting the loop, actual implementation of the MAST model might be able to hide these burst memory transfer with an optimizing compiler. Since it is not difficult for the compiler to find out when a loop starts, those bursty

stores and loads can be scheduled so that the memory transfer can start much earlier than the loop execution, and is overlapped with instruction execution[5].

To analyze the MAST model on loops using this techniques for various value ranges of *loop iteration numbers*, γ, there are three cases[6]:

Case 1: $\gamma < 1$

The loop body is a conditionally executed section of code, as though it were placed within an *if* statement. The analysis would treat it exactly as such, effectively ignoring the backward branch at the end of the loop body.

Case 2: $\gamma = 1$

The loop body is treated as straight line code with respect to the code surrounding it. Effectively, both forward and backward branches in the loop are ignored.

Case 3: $\gamma \geq 2$

Except for the first and last iterations, the loop body should begin and end with the same cache state (in so much as the cache state can be precisely known). The cost of the loop execution is therefore the sum of the cost to bring the cache from the pre-loop state into that cache state plus the cost of the loop body execution times γ-2 times plus the cost to bring the cache into the desired post-loop state.

4.3.4 Analysis of MAST Model in Loops

Complete analysis of the MAST model in loops to obtain a closed form for cache performance might be too complicated. Further, loop nesting makes the analysis even more difficult.

Instead, the performance analysis of the MAST model in loops here is based on an instruction sequence in a simple loop with no nesting — an analysis similar to Section 4.3.2. It is a steady state analysis (i.e. the loop

---

[5] Note that this cache prefetching is much more efficient than the current cache prefetch schemes because the memory load and store can be scheduled much earlier than it is possibly needed.

[6] The above does not consider the situation when $1 < \gamma < 2$. A possible solution, which is unfortunately computationally rather complex, is to treat it as both $\gamma = 1$ and $\gamma = 2$ and to weigh the values to approximate the actual value of γ. A reasonable approximation is to simply consider the case $\gamma = 2$.

body has been iterated many times already) and the reference is sequential, with one reference per line.

Given a cache of size κ and a reference program size λ, the cache performance in simple loop using the MAST model can be summarized as follows:

- $\lambda \leq \kappa$, fully associative cache

  The steady state cache line hit ratio for the MAST model is 1 — same as those from LRU, FIFO, and random. This is because the whole loop can be put in cache.

- $\lambda > \kappa$, fully associative cache

  The steady state line hit ratio is κ/λ. This performance is even better than that from Belady's MIN algorithm, which is usually considered as "the unrealistic optimal replacement" algorithm. The interesting point is that the MAST model in this case is extremely easy to implement. During the first iteration, the cache is filled with part of the loop body. Once the cache is filled, its content is "frozen" — cache content is kept unchanged until the last iteration of the loop. Any reference that is not in cache is made directly from main memory, bypassing the cache.

- direct mapped cache

  With a direct mapped cache,

  - if $\lambda < \kappa$, the line hit ratio is 1;
  - if $\kappa < \lambda$, the line hit ratio is κ/λ.

  This cache performance is again better than those from any of current cache management schemes using direct mapped cache. Further, the cache line hit ratio will never be 0 using the MAST model in loops whereas this is true for current direct-mapped cache design with cache size larger than half the loop size.

One important observation from this analysis is that cache performance improvement in loops using the MAST model increases *monotonically* with the cache size *even if the cache size is very small (e.g. 4 words).* However, in current cache designs using LRU replacement policy, cache improves system performance in loops *only if* it is larger than a certain cache size.

4.4 Divergence-of-Flow

A cache state is said to be **diverged** if there are more than one possible cache state which the current cache state can enter and the cache state of divergence is unknown at compile-time. Figure 4.6 shows a control flow graph for divergence-of-flow. The divergence-of-flow is typically caused by a conditional branch — an example is if...then...else. Note that conditional branches which implement loops are detected as a special case and are handled as described in the last section. For unconditional branches such as goto and jump statements, the cache state of divergence can be determined at compile-time, and they do not belong to this class.

In the MAST model, this divergence-of-flow may be treated in two different ways:

- trace allocation,
- probabilistic allocation.

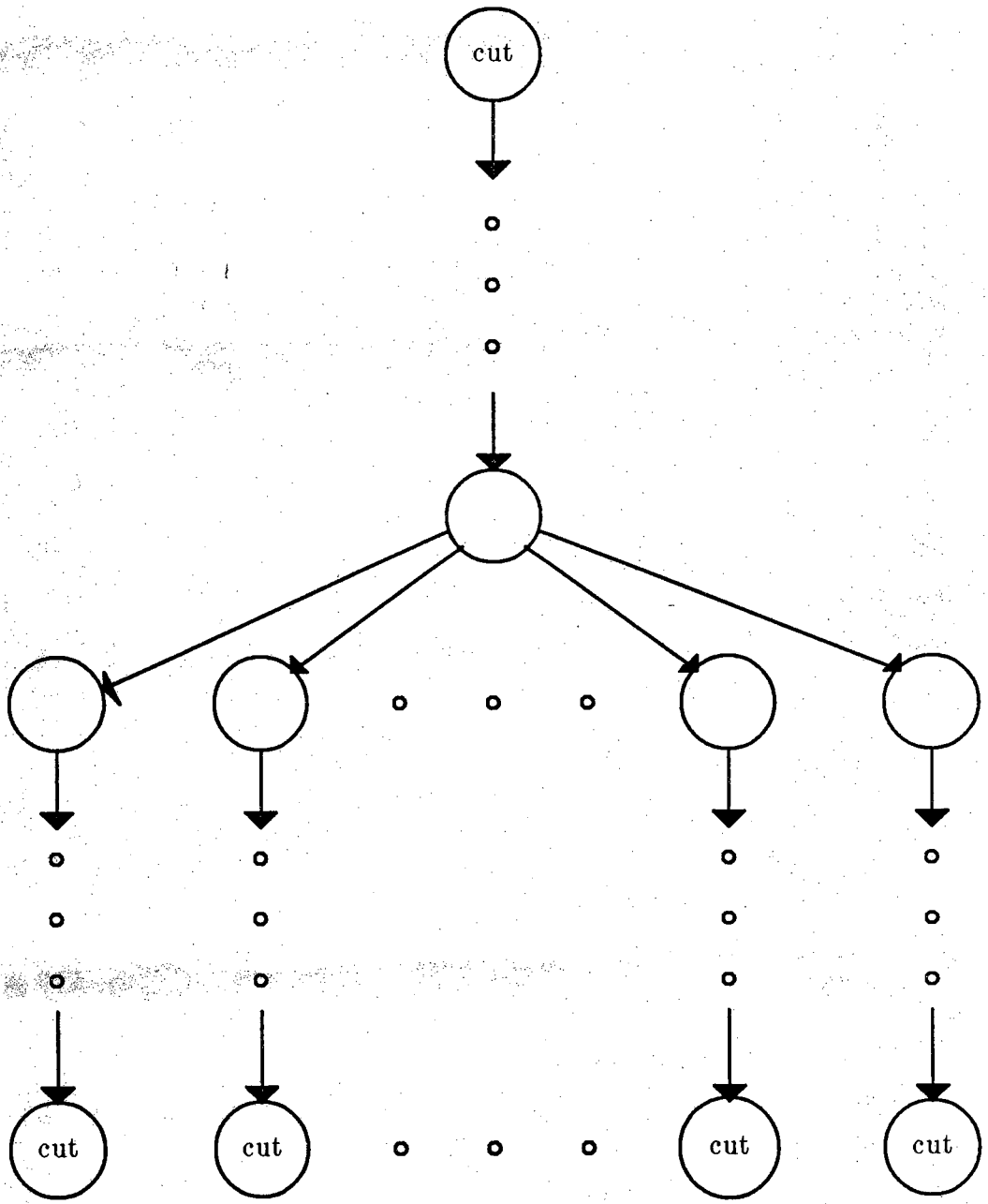


Figure 4.6: Divergence-of-Flow

4.4.1 Trace Allocation

A **trace** is defined as a control flow path from some stage α to stage β in the program flow graph of a reference program, where stage α is the ancestor of stage β [Fis81] [Ell85]. In the MAST model using trace allocation, a cache control sequence for a reference program is found as follows. First, a program control flow graph and data dependency analysis are used at compile-time to do branch prediction — to predict and assign branching probabilities to each of the branching targets [Smi78a] [Smi78b] [McH86] [Lil88] [LeS84]. Then, the trace which is most likely to be taken is selected. This trace is basically a large basic block which spans the code from the last MAST cache cut point before the divergence to the first MAST cache cut point after the divergence on the chosen flow path.

By analyzing the MAST model on this trace, the cache contents at the point of divergence are found (assuming the chosen path is the one actually taken); hence, this stage becomes a MAST cache cut point for other traces. All other diverging paths may be analyzed starting at that MAST cache cut point with the cache contents defined for that stage, as is shown in Figure 4.7.

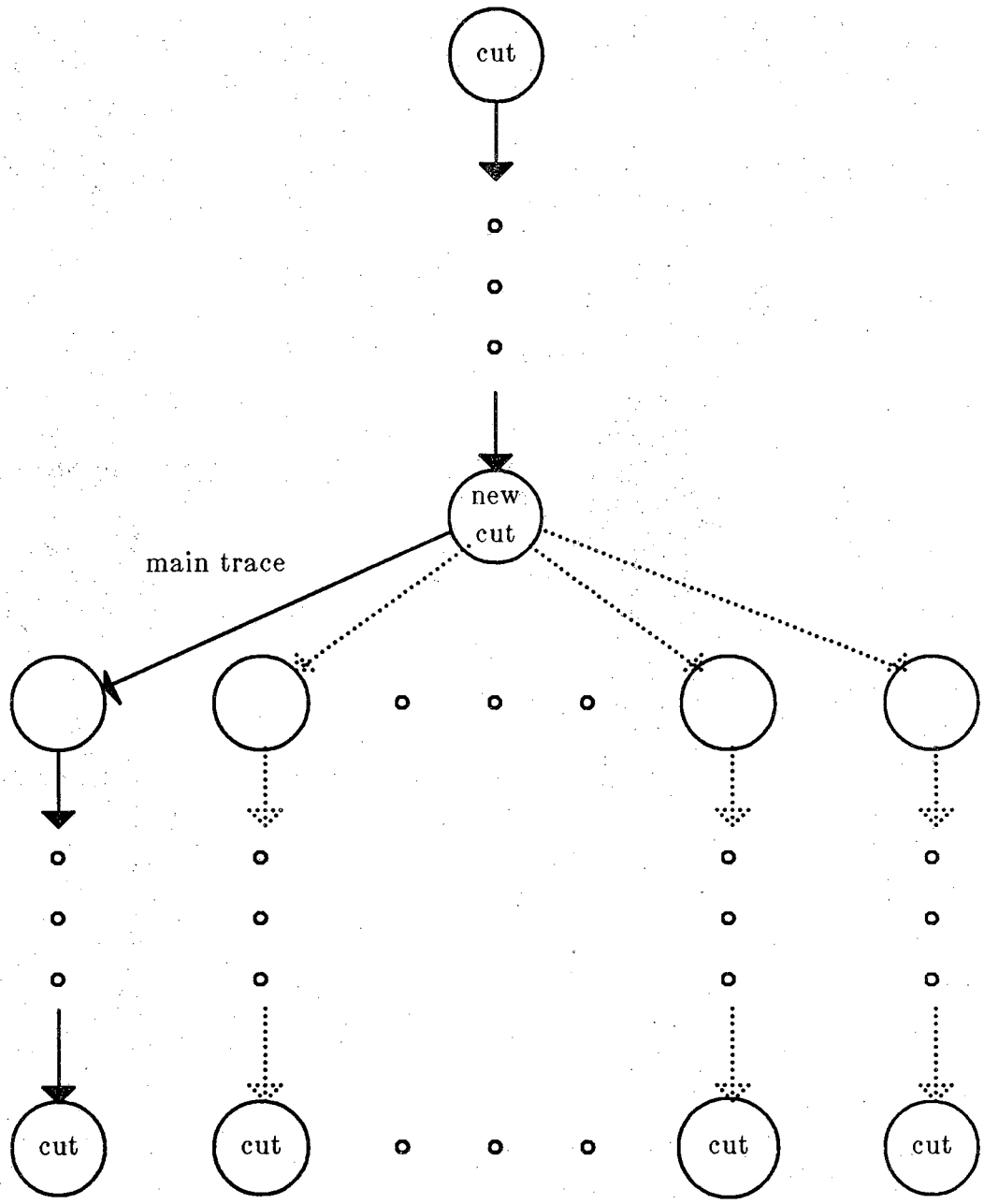


Figure 4.7: Trace Allocation for Branches

Previous research on program reference behavior found that branching probabilities of divergence-of-flow (and convergence-of-flow) are rarely distributed evenly — it tends to flow along one path much more often than the others [DiM87] [LeS84] [McH86]. Using the run-time profile of program execution, the branching probability at the divergence-of-flow can be estimated and is used in the "branch prediction". The accuracy of this branch prediction is quite high, usually between 80 percent to 90 percent. Techniques in program flow analysis and compiler optimization might also be useful in predicting which trace the MAST model should pick up [Die87].

However, it might happen that the branching probabilities to any of the divergence-of-flow paths are about the same or the branching probabilities of these paths are completely unknown statically. In these cases, any of these paths are equally good and the MAST model can randomly pick up any trace as its main path. In the MAST model, this also means that branching probabilities with small differences in values can be explicitly assigned to choose a particular branch as its main trace. For example, if a two way branch with equal branching probabilities is considered, branching probabilities of 0.49 and 0.51 might be assigned to each of the two paths respectively (instead of 0.5 and 0.5). This is very similar to the approach that Trace machines from Multiflow Inc. use to pick traces.

The concept of "traces" is very common in microcode compaction [Fis81] and automatic parallelization for VLIW machines [Ell85]. However, there is one big difference between traditional trace scheduling techniques and the trace allocation described here. In traditional trace scheduling, the most difficult part is to maintain program correctness and is not the trace selection. Hence, sophisticated program transformation is necessary.

In cache allocation using the MAST model, no program correctness need to be maintained. If the MAST model picks one trace while the program executes another trace, the only penalty is the small overhead of **cache cold start** — information in cache is no longer useful and the cache needs to fill up with useful lines again (i.e., cache miss). Hence, the trace allocation used by the MAST model is much simpler.

In current cache replacement schemes like LRU and random access, this penalty of cache cold start also exists when a divergence-of-flow is encountered. When a branch occurs, the working set of references changes and the cache contents might not be useful to the new working set environment.

Using trace allocation to handle divergence-of-flow in the MAST model has the advantage of dividing a reference program into many small program traces, each of which is typically a basic block. Hence, the complexity of the MAST model is much reduced. This approach is much better than the probabilistic allocation described below; hence it is chosen to be used in the MAST model.

4.4.2 Probabilistic Allocation

Assuming that the probabilities for all diverging paths being taken are known, it should be possible to exhaustively search for the optimal allocation by computing the costs for every possible allocation, with the portions on the diverging paths weighted by the probabilities of the paths [Nem66] [Agr79]. Figure 4.8 shows that probabilistic allocation of the MAST model.

Although the exhaustive search may sound prohibitive, it is possible that cut points are sufficiently frequent so that the length of the reference sequences considered is small; hence, the search becomes feasible.

Compared to the trace allocation, this approach is much more complex. Since the effectiveness of both trace and probabilistic allocations is about the same, trace allocation is chosen for the MAST model. Hence, we do not go into details of the probabilistic allocation. However, the basic idea of this allocation is very similar to some classical control problems and people who are interested in this should refer to [Nem66] and [Agr79].

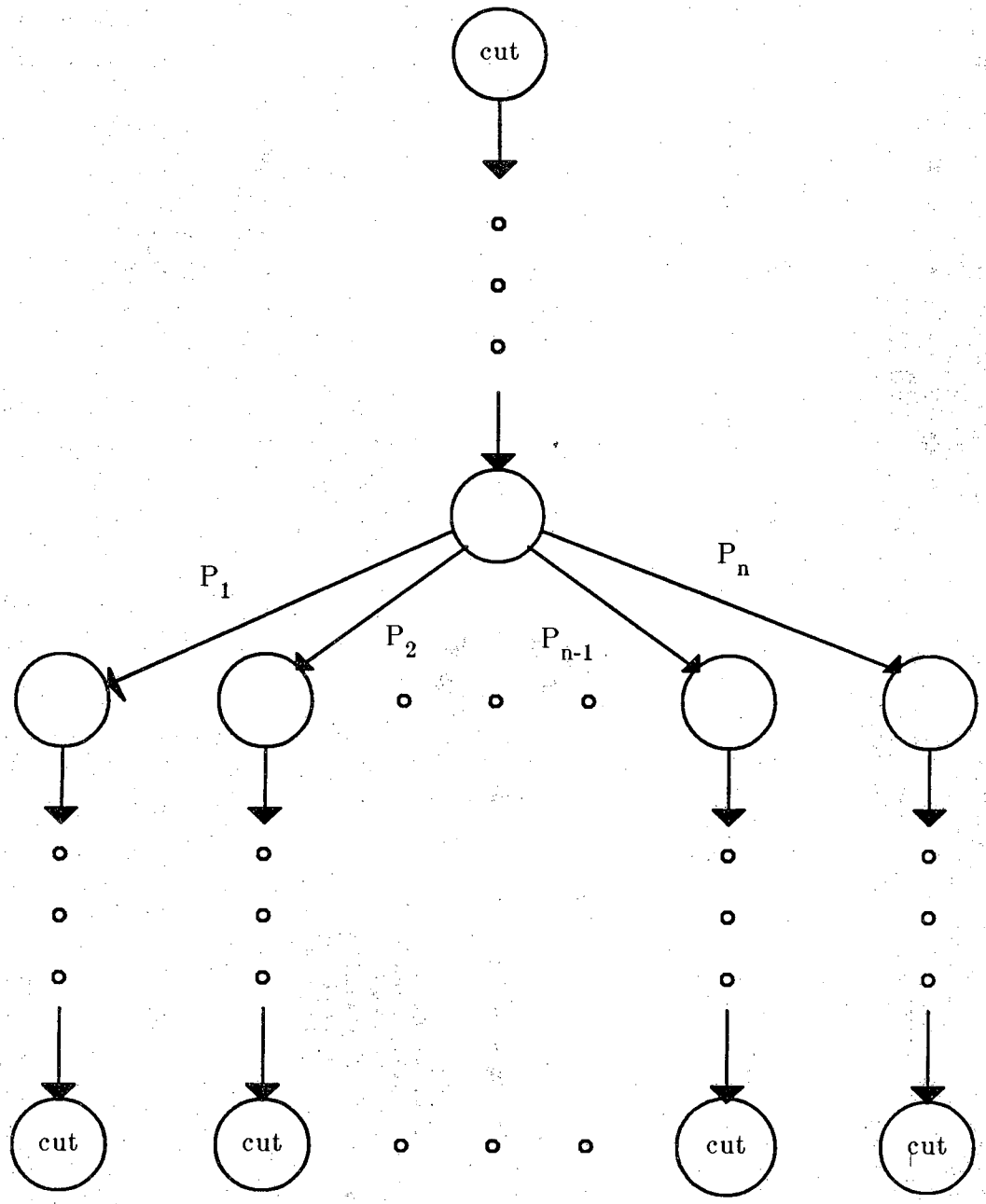


Figure 4.8: Probabilistic Allocation for Branches

4.4.3 Convergence-of-Flow

A cache state is said to **converge** whenever two or more flow paths merge at that stage, as shown in Figure 4.9. A convergence-of-flow typically occurs at labels and at the end of structured control constructs such as *if* or *case*. Although the directed flow graph points to the opposite direction, as compared to the divergence-of-flow, the analysis principle is the same: Hence, we do not repeat the analysis here.

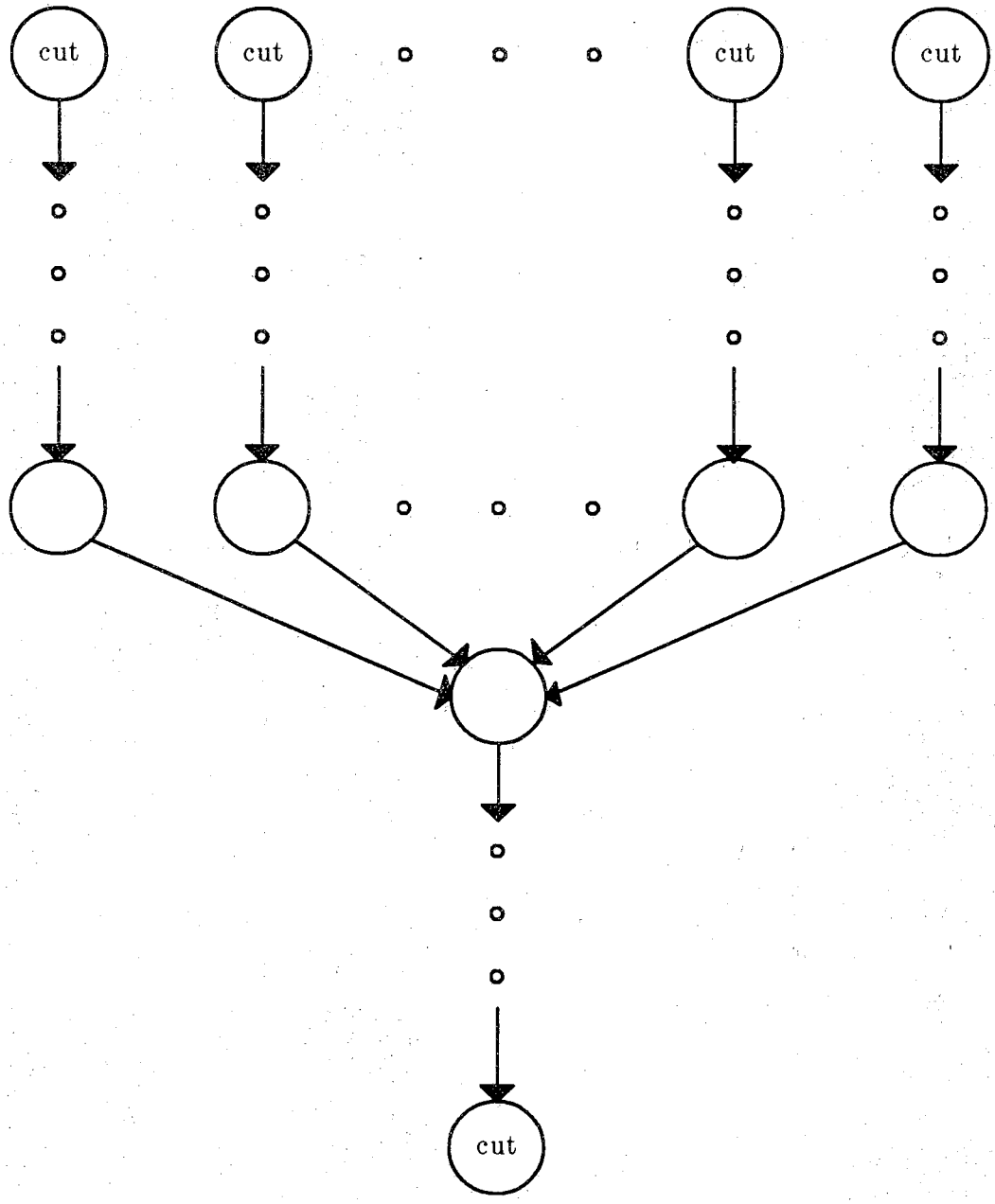


Figure 4.9: Convergence-of-Flow

4.5 Subroutine Calls

Subroutine and function calls are another common control structures in program. It is estimated that about 13 percent of the statically counted and 12 to 45 percent of the dynamically executed High Level Language (HLL) statements are *call and return* [AlW75] [Lun77] [Tan78] [PaS82] [Kat83]. As in loops and divergence and convergence of flow, the exact memory reference sequence within a subroutine call and its effect on the calling routine is unknown at compile-time.

There are at least three basic approaches that the MAST model can use to handle subroutine calls:

- In-line expansion,
- Configuration save, and
- Configuration restart.

Of these three approaches, the first one — in-line expansion — gives the best result whenever it applies. In other cases, the third approach — configuration restart — seems to work best.

4.5.1 In-line Expansion

In-line expansion (also called procedure integration) of a subroutine is the technique in which the compiler analyzes each subroutine call by expanding the called subroutine in-line [Sch77] [AhU77] [AhS86]. That is, a subroutine call is replaced by the body of that subroutine in a program. In the MAST model using in-line expansion, since the resulting memory reference program after the expansion is a subroutine-free reference program, the same techniques and algorithms discussed previously can be used.

Whenever applicable, this yields the best possible performance. This is because cache controls can be different for each of the routine call.

There are several good side-effects of in-line expansion. This technique eliminates the overhead of the subroutine calls: control linkage, register saving and restoring, and parameter passing. Since it effectively increases the size of basic blocks, better global compiler optimization might be obtained. Constants can be propagated, common subexpressions can be found, and better register allocation can be done. It also eliminates two branches (a call and a return) and its branch penalty.

However, there are limitations to this technique. It is clearly not desirable for long subprograms or for routines which invoke other subprograms. This is because it greatly magnifies the length of a reference programs and likewise increases the size of the code generated . In theory, the code space would be exponentially increased by in-line expansion if every subroutine calls more than one subroutine, and is itself called more than once. This situation is even worse when a subroutine is called several times in a loop. Moreover, it is usually very difficult to implement this technique efficiently. In case of recursive calls, it might even be impossible to implement. Separate compilation of subroutines is another problem because without the subroutine body, in-line expansion cannot be performed.

Hence, a more important goal of in-line expansion techniques is to expand subroutines *selectively.* The PL.8 compiler expands leaf procedures and simple functions [Rad83] because their bodies are relatively small compared to the calling overhead. The Stanford UOPT includes a cost-driven in-line expansion phase [Cho83]. Hsu [Hsu87a] suggests limiting in-line expansion only to those execution paths that have a high probability of being taken. Ball [Bal79] describes a technique for predicting the code improvement that can be expected due to constant folding and test elimination when a subroutine call involving constant actual parameters is expanded in-line.

A precise prediction of which subroutines should be expanded would be extremely hard to obtain. However, a simple guideline can be given: do not expand a procedure which is relatively large, has few constant parameters, and is called many times statically.

4.5.2 Configuration Save

Instead of using the complex in-line expansion to handle subroutine calls, another approach that the MAST model can use is the **configuration save.** In this approach, either the caller or called routine is responsible to save any portion of the current cache state which the called routine may change and to restore it on return.

In the MAST model, this approach makes the cache allocation of the caller completely ignorant of the subroutine's existence: cache allocation for the caller uses a reference program which has no representation for a subroutine invocation. Further, after cache allocation in this "routine free

program" is found, the cache state of the routine caller immediately before the subroutine call becomes the MAST cache cut point for the called routine at its entry and exit. Note that the restoration of cache state can be overlapped with instruction execution inside the routine.

Although this approach is easier to implement than the in-line expansion approach, there are still some limitations to its use:

- If the cache size is large, the subroutine body is small, and the frequency of routine call is high, a large fraction of program execution time will be spent in copying the cache state to and from main memory.
- Unnecessary restoration of the cache state might result. If the overhead of restoring cache state is very high and the reference frequency of this information after the subroutine call is very low, it might be cheaper to reference directly from main memory, bypassing the cache.
- Using burst mode to restore the cache state on return might cause the CPU to become idle during cache restoration. This is even worse in multiprocessor environment, where bus traffic is the main performance bottleneck.

Most of these disadvantages are also true for register allocation. Since only a small fraction of information in cache might be used immediately after the routine return, it is desirable to delay the "necessary portion" of cache restoration so that the restoration can be overlapped with program execution. This is the key feature of the next approach — the configuration restart.

4.5.3 Configuration Restart

The **configuration restart** approach to handle subroutine calls in the MAST model is very similar to the configuration save approach. The only exception is that there is no explicit saving and restoration of cache state before and after the routine call. It is the responsibility of the called routine to store back any "dirty" cache cell when that cell is replaced.

In the MAST model, since there is no saving and restoration of cache state before and after the routine call, the cache state is assumed to be empty upon entry and exit of the subroutine. This assumption is justified by the fact that references used in the routine are quite different from those

used by the caller (due to the change of working set).

Using this approach, every occurrence of a routine divides a reference program into three segments:

(1) From the start of a reference program to the stage immediately before the routine call.

(2) The body of the called routine.

(3) From the stage immediately after the routine call to the rest of the program.

Each of these three program segments can be analyzed independently of the others.

Relative to the configuration save approach, this approach yields some surprising benefits:

- Since every occurrence of a subroutine divides a reference program into three pieces, each of which can be analyzed independently, the complexity of the MAST model is greatly reduced.
- Burst mode of information transfer is avoided. Moreover, the restoration of cache after the routine call can be scheduled to overlap with instruction execution before and after the routine is returned.

Clearly, this approach is much better and more practical than the in-line expansion and the configuration save approach. Hence, it is chosen as the strategy to handle subroutines in the MAST model.

4.6 Ambiguous References

The MAST model proposed so far makes no distinction between data and instruction references — the same model can be applied to a reference program containing either or both data and instructions. However, there is a nasty feature of data references which instruction references do not have and which requires special handling in the MAST model. This is the **reference ambiguity of data.**

In the MAST model, it is assumed that the mapping of a name (or label) to the physical memory address is unique and is known at compile-time. However, due to pointers and indirect references of data, this is not always true. For example, given a pointer to an array (e.g. *a[i]*), the name of this pointer might point to different elements of the array at different

stages in a reference program. Moreover, if there is another name *a[j]* referenced in the same program, *a[i]* and *a[j]* might or might not be equal, depending on the values of i and j. Often, this cannot be known at compile-time. This creates problems to the MAST model because:

- Without a unique binding of a name and its physical address, the cache set which an ambiguous name reference might affect might not be known at compile-time. In other words, more than one cache set might need to consider the effect of referencing an ambiguous name in the MAST model.
- It might not be possible to determine accurately the cost of cache state transitions for referencing an ambiguous name α in the MAST model. This is because if there is some datum β in cache which is ambiguously aliased to α, the transition cost of referencing α cannot be determined accurately. It depends on whether α and β are equal.

The problem of ambiguously aliased references in programs has been recognized as one of the hardest problems in compiler optimization. Current management schemes for any level of the memory hierarchy do not have any special handling to this problem. For example, in register allocation, any ambiguously aliased reference bypasses the register file and is referenced directly from main memory[7].

In current cache management schemes, the concept of ambiguous aliased references does not exist. This is because current cache management schemes are purely controlled by hardware and the binding of a name to an address is precisely known dynamically. However, the great benefit obtained from understanding program reference behavior using program flow analysis is sacrificed.

In this section, the MAST model is extended to handle ambiguous aliased references. Types of ambiguous aliased references in cache are first identified. Basic guidelines for any extensions of the MAST model to handle ambiguous aliased references are then suggested. Finally, an example of such an extension is given.

---

[7] In load/store architectures, where every arithmetic or logic operation is a register-to-register operation, any update of an ambiguous name α has to update the main memory immediately and any further reference of names that are ambiguously aliased to α has to be referenced from main memory.

One important observation about the ambiguous reference problem in the MAST model is that *current cache hardware structures cannot handle ambiguous references properly if program flow analysis is part of the cache management scheme.* This may have been due to the fact that when cache was first proposed, the concept of ambiguous aliased references was not well-known. The design goal of cache at that time was to have some memory buffer that was transparent to software and was purely controlled by hardware. Since then, cache organizations and structure have been kept unchanged and the "software transparent" feature is always assumed. This also explains why all cache management schemes involving program analysis [PaG83] [Bre87] are only for instruction cache.

For this reason, the proposed extension of the MAST model to handle ambiguous references in the next section might *not* be very efficient. It is proposed here only to complete the MAST model. In Section 4.7, a new memory hardware, called the **CReg** (pronounced as C-Reg), is proposed as the correct memory component to handle ambiguous aliased references.

4.6.1 Types of Ambiguously Aliased References

To understand the types of ambiguously aliased references in the MAST model, precise definitions of terms **set_map** of a reference, **line_map** of a reference to a cache set, **ambiguous reference**, **set_ambiguity**, **line_ambiguity**, and **cross_ambiguity** are necessary. Note that although some of these terms are also used in register allocations, their meanings might be different.

Definition 4.3: Set_Map of a Name

The **set_map** of a reference α is defined as the set of all cache sets which reference α might map to.

Definition 4.4: Line_Map of a Name in a Cache Set

The **line_map** of a reference α in a cache set λ is defined as the set of all lines in main memory which reference α might map to and all these lines map to the same cache set λ.

For any unambiguous reference β, the size of set_map of β is one and the size of line_map of β in that cache set is also one.

Definition 4.5: Ambiguous Reference

A reference α is said to be an **ambiguous reference** if the size of

either the set_map of reference α or the line_map of reference α to at least one cache set λ is greater than one.

An example of an ambiguous reference in cache management is a pointer α. Suppose pointer α can point to lines *1*, *2*, *3*, *4*, *5*, and *6* in main memory. If all lines with an even line number map to cache set *0* and those with odd line number map to cache set *1*, then the set_map of α is $\{0, 1\}$; the line_map of α in cache set *0* is $\{2, 4, 6\}$; the line_map in α of cache set *1* is $\{1, 3, 5\}$; and pointer α is an ambiguous reference because the size of its set_map is 3.

Based on these definitions, there are at least three situations where ambiguous references cause problems for the MAST model. They are **set_ambiguity, line_ambiguity**, and **cross_ambiguity**.

Definition 4.6: Set_ambiguity

A reference α is said to be **set_ambiguous** if the set_map of α is greater than one.

This set_ambiguity of reference α causes problems to the MAST model because the cache set which reference α affects cannot be known at compile-time.

Definition 4.7: Line_ambiguity

A reference α is said to be **line_ambiguous** in cache set λ if the line_map of α in cache set λ is greater than one.

Definition 4.8: Cross_ambiguity

Two references are said to be **cross_ambiguous** to each other if at least one of them is an ambiguous reference and at least one non-empty line_map of these two references in some cache set λ overlap.

This cross_ambiguity of references creates problems in the MAST model because the transition cost of referencing might not be determined precisely. If two references α and β are cross_ambiguous to each other and β is in cache while α is being referenced, the transition cost of referencing α depends on whether they are equal or not. It they are equal, the transition cost is the cost of referencing from cache. On the other hand, if they are not equal, the transition is the cost to reference from main memory.

While the concept of cross_ambiguity of references has been known in compiler optimization for a long time, the other types of reference ambiguities are complete new to cache design. Note that parameters of cache organizations such as cache line size can partially determine the

amount of reference ambiguity in cache. For example, with a larger line size, the size of the line_map of references might be smaller.

4.6.2 MAST model for Ambiguous References

The extension of the MAST model to handle ambiguous references can be considered as providing answers to the following questions:

Question 1:

If the set_map of an ambiguous reference α is greater than one, which cache set in the set_map of α should be used to determine whether the reference should bypass the cache? This question is extremely important because the decision of "through-cache" or "bypass-cache" of an ambiguous reference need to be consistent in the MAST model analysis for different cache sets.

Answer:

For each $i \in$ set_map of α, the size of line_map of α in cache set i is found. The cache set κ in which the size of the line_map of α is the largest should be chosen to determine the "through-cache"/"bypass-cache" decision of referencing α in the analysis of cache sets in all its set_map. This is because the probability of having α belong to cache set κ is the highest.

Question 2:

Since the MAST model analysis for one cache set might determine the "through-cache" or "bypass-cache" decisions of some ambiguous references in other cache set, deadlock situation might occurs. This is because the MAST model analysis of each cache set might wait for decisions from the analysis of other cache sets and none of them would proceed its own. How should the MAST model handle this situation?

Answer:

The cache set with the least number of pending (or waiting) decisions should be chosen and all the pending decisions can be assumed to be cache-bypassed. The cache-bypass decision is assumed because it is usually not beneficial to place an ambiguous datum in cache.

Question 3:

In the MAST model analysis of one cache set, it might happen that when an ambiguous reference α is being made, there are some

references in cache that are cross_ambiguous to α. Under this situation, the transition cost of referencing α cannot be determined precisely. How should the MAST model handle this?

Answer:

First, each of the cross_ambiguous names should be assumed to be distinct. This is because it is better to leave holes in cache than to face the problem of insufficient cache space for those references that are placed in cache according to compile-time decisions. Let the set of cross_ambiguous names in cache be S and the probability of having $\alpha \in S$ be $P(\alpha)$. Then, the transition cost of referencing α is:

$$P(\alpha) * Cost_{cache} + (1 - P(\alpha)) * Cost_{oper}$$

where $Cost_{oper}$ might be the cost of referencing directly from main memory, cost of placement, or cost of replacement, depending on the type of state transition of that reference.

Perhaps, an example might help clarify most of these ideas. Suppose that the given reference string is a,b,c,a,b,c; the cache is direct-mapped with size is two; the possible mappings of name a, b, and c to lines in main memory is shown in Table 4.2; all lines with odd number map to cache set *1* and those with even line number to cache set *0*.

Table 4.2: Mapping of Names to Lines in Main Memory

Name	Possible Lines Mapped to Name	Cache Set Used to Determine Cache Bypass
a	*1,2,3*	*1*
b	*1,2,4*	*0*
c	*4,6*	*0*

The last column of Table 4.2 shows the cache set which the MAST model analysis uses to determine the cache-bypass or cache-through decision for each name. The cache-bypass decision for name *a* is determined by the MAST model analysis for cache set *1* because the size of the line_map(α, 1) is the largest. Similarly, the cache-bypass decision for name *b* is determined

by the MAST model for cache set *0*.

Table 4.3: Equality Probabilities for Names

Names	Equality Probabilities
a = b	2/9
a = c	0
b = c	1/6

From Table 4.2, the probabilities for having two names equal can be deduced and is shown in Table 4.3. The references substring for cache set *0* is *a,b,c,a,b,c*; that for cache set *1* is *a,b,a,b*. Deadlock occurs because the MAST model analysis for cache set *0* waits for cache-bypass decision for *a* (determined by the analysis for set *1*) and that for cache set *1* waits for cache-bypass decision of *b* (determined by the analysis for set *0*).

To solve this deadlock problem, any reference to *a* is assumed to bypass the cache. The MAST model analysis for cache set *1* is shown in Figure 4.10. References to name *a* bypass the cache while references to name *b* go through the cache. With these two decisions, the MAST model analysis for cache set *0* is shown in Figure 4.11. Note that the symbol *P(x,y)* in Figure 4.10 and 4.11 denotes the probabilities of having name *x* being equal to name *y*.

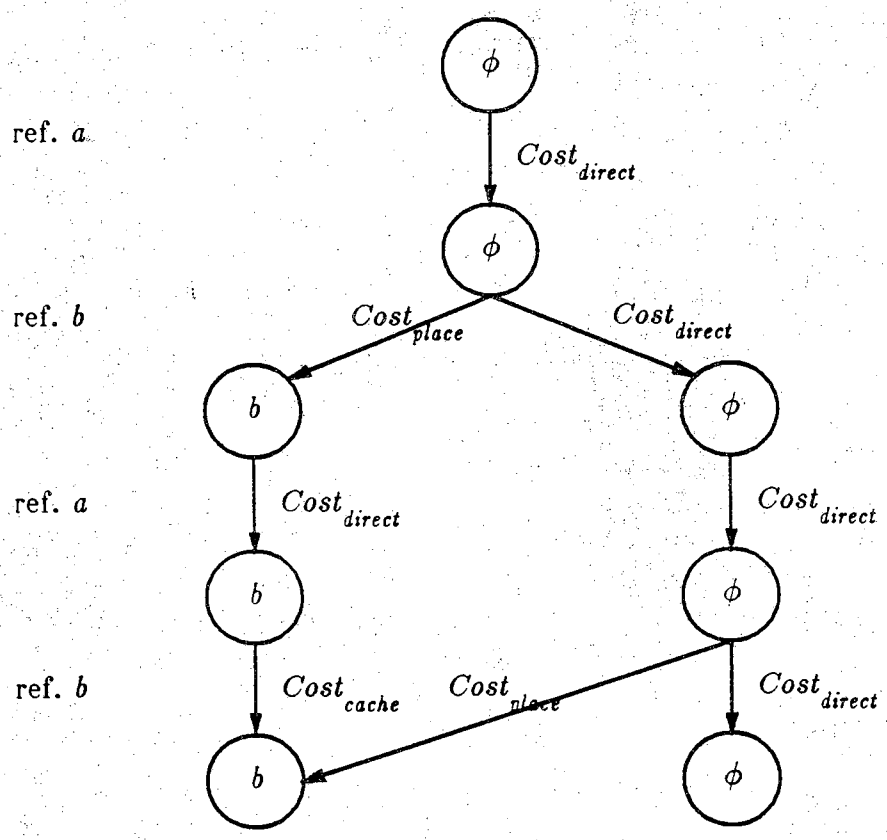


Figure 4.10: MAST Model Analysis for Cache Set *1*

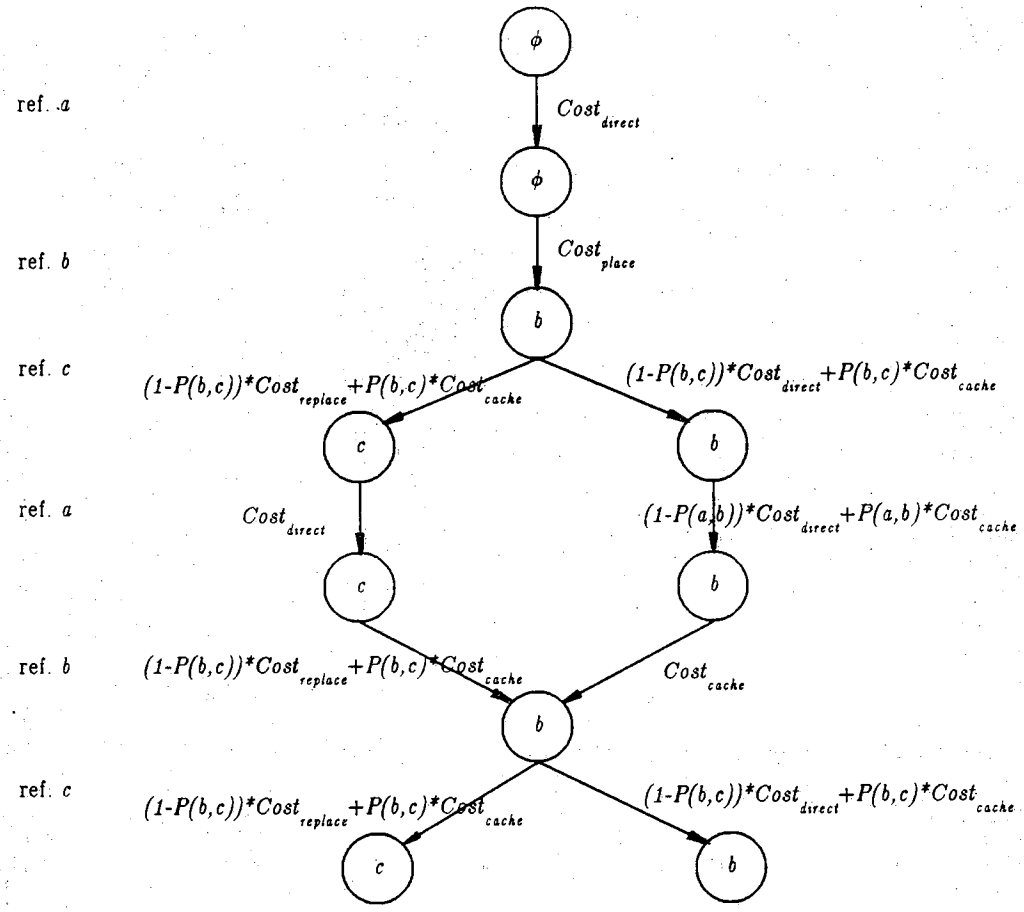


Figure 4.11: MAST Model Analysis for Cache Set *0*

4.7 CRegs

This new hardware is a joint invention between Professor H. Dietz and I during summer 1987. Since the management of CRegs is not directly related to this thesis, only a brief description is given here. Details about the structure, operations and management of CRegs should be referred to [DiC88].

The **CReg** (pronounced "C-Reg") is a hardware structure which combines the structures of cache and registers. Unlike registers, cache-registers (CRegs) may be used to buffer values which may have ambiguous aliases; unlike cache, CRegs provide the ability to use short names for variables instead of addresses (thereby reducing instruction-fetch bandwidth requirements) and also provide for conceptually duplicate entries (many-to-one mapping of short names into addresses). CRegs provide all advantages of registers; but CRegs provide these advantages for all values, ambiguously aliased or not.

The conceptual structure of a CReg memory cell is a superset of both cache and register cell organizations, as illustrated in Figure 4.12.

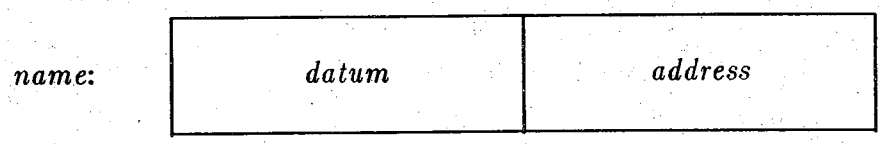


Figure 4.12: CReg Memory Cell Structure

Each CReg is a register which holds both address and data fields. When a CReg is referenced (by CReg number — a short name), an associative search is made to find neighboring CRegs which have the same value in their address field. Any CRegs found by this association *are* aliases for the CReg directly named, and the CReg hardware simply maintains coherence of these entries. This associative function is implemented by hardware very similar to that implementing the associativity of a cache, however, unlike a cache, CRegs avoid making memory references on an aliased *Load* operation by using *duplicate entries in the CReg array.* (The precise operation of CRegs is described in greater detail in the example of the following section.)

Although the STM (generalized Short Term Memory cell) [Sit79] employs a memory cell structure similar to that of a CReg, STMs did not support CReg-like associative function. Likewise, the "rack" described in [StS85], suggests a similar cell structure, but is not associative. The register-addressed stack cache mechanisms of various processors could be

argued to provide benefits similar to that of CRegs; however, they do so only for items in the top few stack cells. Since these items are a subset of the items which could have been kept in registers, stack caches also fail to provide a solution to the aliased-item reference problem.

The implementation, and hence the circuit complexity, of a CReg array is virtually identical to that of a similar-size cache; the CReg array is slightly simpler because the hardware is explicitly told where to make each entry (what CReg number), whereas conventional caches use hardware-implemented policies, such as LRU, to decide which line to replace. However, CRegs can be managed as efficiently as registers, hence, unlike cache, very small CReg arrays are quite useful. For example, simply replacing the registers of a conventional processor with CRegs (and, incidentally, not even changing the instruction set) is typically both feasible and effective to the extent that the number of memory references made by the processor can often be halved.

4.7.1 An Example

An example clearly illustrates the advantage of being able to use CRegs for all value references, whether ambiguously aliased or not. Consider the FORTRAN code in Figure 4.13.

```
C
C     A call-by-address subroutine
C
      SUBROUTINE NASTY(I, J, K)
10    I = J * K
20    K = J + K
      RETURN
      END
```

Figure 4.13: FORTRAN Sample Code

The subroutine *NASTY* operates on three arguments, *I*, *J*, and *K*, which are all passed using call by address. Since FORTRAN permits *I*, *J*, and *K* to reference the same cell of main memory, the values of *J* and *K* cannot be blindly placed in conventional registers in the code for line *10* and simply reused in the code for line *20* — to do so would produce incorrect results if *I* refers to the same main memory cell as either *J* or *K*. To place these variables in conventional registers, the compiler would need to know precisely which of the arguments referenced the same cells — the classic ambiguous alias problem discussed earlier. Hence, unless the compiler is permitted to generate multiple encodings of *NASTY* (one for each possible set of variable aliases), none of *I*, *J*, and *K* can be kept in registers.

However, all of these variables can be placed in CRegs. In fact, if this is done, the references in line *20* will *always* be served within CRegs — main memory will not be accessed. Table 4.4 shows all possible combinations of aliases for *I*, *J*, and *K*, and, for each alias set, where the values of *J* and *K* are to be found.

Table 4.4: CReg Place of Reference for *J* and *K*

Aliases	Where *J* is	Where *K* is
I, *J*, *K* disjoint	CReg for *J*	CReg for *K*
I is *J*	Assoc. *I,J*	CReg for *K*
I is *K*	CReg for *J*	Assoc. *I,K*
J is *K*	Assoc. *J,K*	Assoc. *J,K*
I is *J* is *K*	Assoc. *I,J,K*	Assoc. *I,J,K*

As indicated earlier, the read requests in line *20* for *J* and *K* are always satisfied within the CReg array. The entries in the table which say "CReg for" are simple CRegs acting as ordinary registers (with no associative access); the entries which say "Assoc." are satisfied by the associative memory function of the CRegs. (Notice that the associative function depends on the existence of *duplicate entries* in the CReg array — which

would not be permitted in a conventional cache.) Further, since all three variables can be placed in CRegs, the references in line *20* would simply use the CReg names, rather than memory addresses, because a CReg name *implies* a main memory address. This fact also permits *entire instructions to disappear*: *Store* instructions can be implicit, using a "dirty bit" and a "lazy store" mechanism.

It can be argued that the references in line *20 might* be satisfied in a more conventional cache, thereby avoiding a main memory reference in the same way that CRegs avoid the reference; however, only CRegs can guarantee that this occurs under all circumstances (as we detailed in the previous section). Even accepting that a conventional cache *might* avert the main memory references as CRegs do, CRegs also reduce the *instruction fetch* bandwidth requirements by permitting short CReg names to be used for *I*, *J*, and *K* rather than long main memory addresses.

4.7.2 CReg Allocation

Given that the source program has been analyzed and that the collection of alias sets [Coo84] [Cou86] [Die87] [[NeP87] is known, the CReg allocation is to assign values to CRegs and to generate code reflecting that assignment. Since CRegs closely resemble registers, it is not surprising that the allocation schemes for CRegs closely resemble those for register allocation, except for the need to operate on alias sets. If, for example, all alias sets obtained from a program are singleton sets, CReg allocation *is precisely* register allocation.

Due to limitations of hardware circuit complexity, the (simultaneous) associativity of a CReg array is constrained to be a small number: typically four or eight (just like the associativity of cache). However, it is quite reasonable to have an array much larger than just four or eight CRegs — breaking the CReg array into sets with smaller associativity. Consequently, the first and the most important rule of CReg allocation is to put all elements from each particular alias set into the same CReg associative set. At first, this sounds overly constrained, since an alias set containing more than four elements cannot possibly "fit" into a four-element CReg associative set, however, experience with compiler automatic parallelization technology [Ste86] has shown that the average number of *simultaneously active* ("live") names within an alias set is very rarely more than three[8].

[8] This number makes sense in that code like a[i]=a[j]*a[k] only uses three names — the number three seems to be a side-effect of the dominance of binary operations.

Another key issue in CReg allocation is CReg **spilling.** An item is spilled from a register if a register is needed for some other item, yet no registers are empty. Here, the problem is that if a single name from an alias set is to be referenced from a CReg associative set other than that which contains the other elements of the alias set, *all* elements of the alias set must first be flushed from the CRegs. This makes spilling of alias sets highly undesirable: spills defeat the benefits gained from CReg hardware automatically maintaining consistency across multiple names in an alias set.

4.8 Pruning Techniques for MAST Model

After the MAST model and its extensions to various program structures are presented, it is appropriate to discuss various pruning techniques to speed up the analysis of the MAST model.

In Chapter 3, it is shown that the computational complexity of the MAST model analysis is of the order $O(n^2m^{2K})$, where n is the total number of rerferences, m is the number of live values at some cache stage, and K is the set associativity. This complexity is relatively low compared to most current algorithms in compiler optimizations; hence its implementation is clearly practical. Various pruning techniques exist to reduce the average computational complexity to "almost linear".

Pruning techniques for the MAST model analysis might come from one of these three sources:

- architecture constraints,
- program behavior, and
- heuristics for the MAST model.

4.8.1 Pruning Techniques from Architecture Constraints

Due to implicit constraints on how a cache should operate, the number of possible cache state transitions is always limited. Although these pruning techniques can be considered as "free", they are quite effective in reducing the computational complexity of the MAST model:

- Cache set size

  Cache set size defines the number of lines in cache which lines in main memory might map to. This also determines the number of possible

cache state transitions (i.e. number of transition arcs) and the number of possible cache states in a cache stage. Since the number of feasible cache states and arcs in a cache stage is an exponential function of the cache set size, small cache set size, which is commonly used in current cache designs, can greatly reduced the computational complexity of the MAST model. This is true even though there are more sub-programs needed to be considered.

- Cache line size

  Cache line size is one of the few main factors to determine the number of distinct live lines in a cache stage. The larger the line size is, the smaller is the number of distinct live lines in a cache stage, hence, the lower is the computational complexity.

- Through cache vs. bypass cache

  The option of cache bypass is added to the MAST mode because it is found that a large fraction of cache performance loss is due to non-beneficial cache demand. However, in current cache designs where the cache bypass option is not available, the computational complexity of the MAST model is reduced. This is because cache is likely to be filled up quickly and any cache states that are partially filled might not be feasible. For each cache state, the number of possible state transition is also one less if no cache bypass is allowed.

4.8.2 Pruning Techniques from Program Behavior

In a reference program, there are several features of program behavior which are very effective in reducing the computational complexity of the MAST model. They can be summarized as follows:

- Static vs dynamic code size

  Cache management by the MAST model analysis is determined at compile-time; hence static code size is used in the analysis. Since the dynamic code size is usually 1000 times or more larger than the static code size, the computational complexity of the MAST model analysis is within a practical bound[9].

[9] This can be compared to current cache designs, in which cache simulations are done using execution traces containing millions of references.

- Liveness of references

  The concept of "liveness" of references in cache greatly reduces the computational complexity of the MAST model. It changes the number of feasible cache states from an exponential function of the number of distinct lines in a reference program to a exponential function of the number of simultaneous live cache lines at any cache stage. Since the number of simultaneous live cache lines is approximately a small constant, the number of feasible cache states is greatly reduced and so is the MAST model complexity. Further, the compiler can also control the number of possible simultaneous live values in programs by live range spilling [Cho83] [ChH84].

- Unambiguous vs ambiguous

  From the compiler viewpoint, the functional difference between register and cache is that cache can hold both ambiguous and unambiguous references while registers can only store unambiguous references. Hence, redundant storage of the same datum in both register and cache occurs in current cache management schemes. One effective way to avoid this is to put ambiguous data references and instruction references in cache and to put unambiguous references in registers. This is called the **unified memory management model** and will be discussed in detail in Chapter 7. With this unified model, the number of references analyzed by the MAST model is greatly reduced, and so is the computational complexity of the MAST model.

4.8.3 Pruning Techniques from Heuristics

In the last two sections, various pruning techniques in the MAST model are presented. These techniques are mainly due to some "external" constraints — cache organizations and program behaviors. In addition to these techniques, there are properties of the MAST model which further reduce the computational complexity of the MAST model. Some of the pruning techniques proposed in this section is quite similar to those proposed for (index) register allocation [HoK66] [Luc67] [Ken71a] [Ken71b] [Hsu85] [Hsu87a] [Hsu87b].

Properties of the MAST model which reduces its computational complexity can be summarized as follows:

- MAST cache cut point

  The existence of MAST cache cut points in program reduces the MAST model analysis because it divides a references program into smaller segments, each of which can be analyzed independently of the others. As is discussed in this chapter, the existence of MAST cache cut points might be due to:

  - loops,
  - subroutines (using configuration restart),
  - branches (using trace allocation),
  - Others such as a cache state with no live lines.

  Since these are common control flow structures found in programs, the MAST cache cut points are very effective in speeding up the MAST model analysis. Moreover, the introduction of artificial cache cut points can further lower the computational complexity.

- Stage cleansing rule

  Let $TotalCost(\kappa,\lambda)$ is the total cache transition cost from stage 0 to stage λ with cache configuration κ at stage λ and $Trans(\lambda,\kappa)$ is the minimal transition cost from cache configuration λ to cache configuration κ. At any cache stage λ, if there exist two cache states α and Q such that

  $$TotalCost(\alpha,\lambda) \geq TotalCost(\beta,\lambda) + Trans(\beta,\alpha)$$

  Then cache stage α can be eliminated from the MAST graph. This is because cache state α can be reached cheaper by going through cache stage β and then α instead of going to α directly.

- Alpha-beta pruning

  This technique is commonly used in search algorithms. The idea is that if the total cost for generating a portion of one path μ is larger than the total cost for generating a particular path, any path containing the portion μ is definitely not the path with lowest cost and can be eliminated from the consideration.

- Heuristics

  Various heuristics can easily be found and used to reduce the the graph analysis time.

4.9 Implementation Techniques for MAST model

Cache control information about each cache state transition after the MAST model analysis might be collected as follows:

- Reference Memory Component

  It indicates where the reference should be made. It might be in cache, or directly from main memory (or elsewhere, if there is other memory component that can be directly accessed by the CPU). If there is γ such memory components, $log(\gamma)$ bits are needed.

- Line Replacement

  This indicates which line in cache is going to be replaced by the currently fetched line (if there is any). The number of bits needed to store this information is $log(\delta)$, where δ is the set size of the cache. A very interesting situation is when $\delta = 1$. This is the case for direct-mapped cache and *no bit is required* since there is no replacement choice in direct-mapped cache.

The MAST model requires that the above cache control information be inserted into the code generated by the compiler. These inserted "cache directives" can be embedded in the generated code in two different forms: explicit cache control instructions or tagging references at the end of each instruction.

The first method is to define explicit cache control instructions for fetching and storing lines between cache and main memory. These explicit cache control instructions are inserted into the program by the compiler. Explicit cache control instruction like those in Table 4.5 might be useful.

Table 4.5: Explicit Cache Control Instructions

Opcode	operand 1	operand 2
CLoad	Cache line number	memory line address
CStore	Cache line number	Cache set number

This *CLoad* instruction is to fetch the contents of the memory location with address given by operand 2 and place it in location given by operand 1 in the cache set in which it belongs. The *CStore* instruction is to store the contents of the location given by operand 1 in the cache set given by operand 2 to the main memory.

The cost of this implementation is the extra execution time needed to execute these explicit cache control instructions. Improvement of cache performance can still be gained in the presence of this overhead because:

- The cost of issuing an explicit cache instruction is always smaller than the penalty for a cache miss.
- Every explicit cache instruction corresponds to a cache miss.

Another main advantage of this technique is that it can be implemented as a coprocessor — *permitting its use by existing, conventional, processors.* In fact, even without any specialized hardware, some benefit can be gained in multiprocessor environments by using a software-simulated cache and block transfers between global and local memory spaces (this is feasible only because global memory references may have extremely long access times).

The other alternative, which is more appropriate in custom-designed processors, is to place the cache control directives (from the compiler) within each instruction, by using a cache-directive tag field. This trades the time to execute coprocessor-instruction cache directives for the need to "borrow" instruction bits (in every instruction which could cause a reference) for cache directives. Although the need for extra instruction bits may increase the instruction word length (and hence the cycle time), the fact that only a couple of bits are typically needed leads us to predict that an extension of instruction word length is not necessary.

A combination of these two methods is also possible. Cache directives might be tagged to an instruction whenever possible. When the address of the fetched operand is too long to be tagged to an instruction, explicit cache control instructions can be used. For example, it is more difficult to embed prefetch cache directives within each instruction than to make separate prefetch instructions, since the prefetch offset may require a large number of bits for its representation. Hence, explicit cache control instructions would be used for prefetch. This minimizes the number of cache control instructions in the execution stream and, at the same time, solves the problem of large offsets in prefetch references.

The MAST model can also be implemented using demand fetch, prefetch, or a combination of the two; in many cases, the distinction between the two is quite vague. For example, a delayed load instruction (as found in most RISC processors) can be considered to be either demand fetch — because it requests data on demand relative to the memory system outside the processor) — or prefetch — because the request is separated from the use within the processor. In either demand fetch or prefetch, the same delays are encountered and the compiler must schedule the fetch/cache activities to minimize idle time: only the positioning of the control "directives" within the execution stream is different. For example, the use of *NOP* instructions to fill-in the gap between issuing and completing a delayed load in RISC processors is not fundamentally different from the (much older) techniques which do not advertise a delayed load but none-the-less allow loads to take several cycles while each instruction waits for a hardware "valid" tag before it uses the content of a register.

4.10 Conclusions

In this chapter, generative uses of the MAST model in various program structures are discussed. It is shown that the foundation of all these extensions of the MAST model lies on the concept of the MAST cache cut point.

When a reference program is inside a loop, the cache configuration is dominated by references of the loop. Hence, cache controls within a loop can be analyzed independently and the reference program can be considered to be three segments — the segment before loop entry, loop body and the segment after loop exit — each of which can be analyzed independently of the others.

For convergence and divergence of flow, trace allocation seems to be the most straightforward and effective method in the MAST model. When subroutine calls are encountered, configuration restart should be employed in the MAST model because a MAST cache cut point is created and no unnecessary saving and restoring of cache configuration is needed.

This chapter also discusses the handling of ambiguous aliased references in the MAST model. It is shown that current cache structure are not capable of handling ambiguous aliased references efficiently if software cache management is part of the cache control scheme. A new memory structure, called the CReg, is proposed as a hardware solution to the ambiguous

reference problem.

Various pruning techniques are proposed in this chapter to reduce the computational complexity of the MAST model analysis. Finally, the implementation considerations of the MAST model are given.

In the next chapter, we will turn to current cache designs and discuss how the MAST model can be applied to these designs with minimum changes.

CHAPTER V
GENERATIVE USES OF MAST MODEL WITH TRADITIONAL CACHE STRUCTURES

5.1 Introduction

In Chapter 3, a new cache management model, called the MAST model, was proposed as an integration of software and hardware approach to provide optimal (or nearly optimal) cache management. Extensions of this MAST model to various program structures were made in Chapter 4. At the end of Chapter 4, various pruning techniques were also suggested to reduce the computational complexity of the MAST model and the implementation technique of this model was given.

It is obvious that cache performance using the MAST model is superior to current cache management schemes — a large fraction of performance loss in current cache management schemes can possibly be regained by using this model. However, there are difficulties in applying this model to current cache design. These are mainly due to:

- Existing architectures

  Machines which are already built or are being built might not have enough flexibility in hardware and architecture (including the instruction set definition) to implement the complete MAST model. The instruction set might have been defined so that only a few (if any) explicit cache control instructions can be added or can be defined by system programmers using user-defined instructions. In other situations, cache control policies such as replacement policy might be fixed and no cache bypass option allowed.

- Instruction word length

  In any system design, the instruction word length is always limited to a small fixed number (e.g. 16 or 32 bits) and bits of an instruction are always very expensive. Hence, for existing architectures, the number of unreserved bits per instruction that can possibly be used for cache control is very few. For example, in the Precision Architecture by

Hewlett-Packard, there is only two unreserved bits per instruction available [Hol88]. As a result, the complete MAST model for explicit cache management might require more cache control bits per instruction than a machine has available.

- Complexity of the MAST Model

  With various pruning techniques for speeding up the MAST analysis (as is discussed in the last chapter), the average computational complexity of the MAST model is similar to (if not lower than) many compiler optimization algorithms and is low enough to be implemented. However, it seems that this complexity is still too high for hardware cache designers and linear time algorithm are preferred.

To overcome these problems and apply techniques in the MAST model to current cache designs, some "simplified versions" of the MAST model are needed. These simplified versions of the MAST model should satisfy most of the following criteria:

- Linear time

  The computational complexity of these simplified versions should be as low as possible — it should be a linear algorithm if possible.

- Minimal hardware change

  The additional hardware required or the cache hardware change should be small. For example, the number of additional cache control bits per instruction required should be only one or two. This allows existing cache hardware to be modified more easily.

- Good solution

  The optimality of cache performance is not the main concern in these simplied versions of the MAST model. If very large effort is needed to maintain the optimality of cache performance while "good or nearly optimal"[1] solutions are easy to obtain, the optimality should be sacrificed.

- Minimal cache control change

  The modification of the cache control policies, such as replacement policy and updating policy, should be limited. Like the minimal hardware change discussed above, the smaller is the difference between

[1] The term "good or nearly-optimal" is very difficult to define. As a rough guideline, the difference between an optimal and a good solution should only be a few percent.

the MAST model and current cache control policies, the easier is the modification of these existing cache control policies.

In summary, cache designers should be interested in simplied versions of the MAST model, which should have linear-time execution, require little change in both cache hardware and software, and have cache performance comparable to the optimal solution obtained from the complete MAST model analysis. This is the main focus of this chapter — the generative use of of the MAST model in current cache designs and the derivation of linear-time, simplified versions of the MAST model.

In Section 5.2, differences between the MAST model and current cache management schemes are identified. Understanding these differences is extremely important because it provides hints on how current cache management schemes should be modified. Section 5.3 discusses the application of live range analysis to current cache management schemes such as LRU, FIFO, random and Belady's MIN algorithm. The cache bypass concept and its use in various cache control schemes is given in Section 5.4. In Section 5.5, a hardware implementation of these liveness and the bypass control bits in cache are presented. The software cache replacement control model is discussed in Section 5.6. Finally, the chapter is summarized in Section 5.7.

5.2 Differences Between MAST Model and Current Cache Designs

Cache design is always one of the few most important design aspects of a computer system. Although numerous research efforts have been devoted to designing and optimizing caches, large improvement of system performance is hard to obtain in current cache design. However, from our simulation study (which will be presented in Chapter 6), we have found that a large fraction of performance loss in current cache designs might be regained by the MAST model. This is mainly due to some fundamental conceptual differences between current cache schemes and the MAST model:

- Cache control

  Should the cache be controlled by pure hardware or by an "integrated approach combining hardware and software"?

- Program understanding

  Should program analysis be included in cache management schemes? In other words, should cache operations be transparent to the user?

- Order of referencing

  Should the reference order of a program be used in cache management schemes?

- Cache utilization

  What is the *right measuring parameter* for cache design? Time or cache hit ratio?

- Cost consideration

  How do the costs and benefits of placing a line in cache influence cache control decisions?

- Reference liveness

  Is information in cache always useful or valid? If not, how can this be determined?

A thorough understanding of these differences between current cache management schemes and the MAST model is necessary. Once these operational differences are identified, current cache designs can easily be modified by selectively incorporating some of these new concepts.

5.2.1 Cache Control

Since cache was invented in the 1960's, it has always been assumed that cache should only be managed by "pure hardware". The rationale behind this assumption is that cache processing must happen as fast, or faster than the CPU executes instructions. If one or more explicit cache control instructions are needed to execute each program instruction, the system would be slowed down. The CPU spends more time executing cache control instructions than executing program instructions (there are one or more memory fetches per instruction execution). Hence, software cache control for individual references is assumed to be infeasible.

Although there are machines which include software cache control in their cache management schemes, this control is used for cache prefetch instead of cache placement/replacement of each individual reference. Some examples are the remote PC in RISC II [PaG83] and the explicit allocate and deallocate cache control instructions in IBM 801 [Rad83].

This assumption, however, is only partially true. Although executing one or more cache control instructions for *every* program instruction execution might seem infeasible, it does not exclude the possibility of using

information about program reference behavior, which is collected at compile-time, in hardware-managed cache schemes. With program flow analysis by the compiler, reference behavior of a reference program can be analyzed statically to determine its best cache allocations. This cache allocation information can then be stored either as explicit cache control instructions or as cache control bits inside an instruction so that cache hardware logic can be used for intelligent cache control.

Explicit cache control instructions can improve cache performance if they are used intelligently. The penalty of a cache miss is always greater than the cost of issuing an instruction, which is usually one cycle in pipeline machines. If the compiler is smart enough to predict correctly that there is going to be a cache miss, an explicit cache control instruction can be used to perform the cache prefetch, thus avoiding a cache miss.

Further, one explicit cache control instruction might control cache allocation of a region (e.g. a few instructions) instead of a single instruction. An example is to define cache control instructions to allocate and deallocate cache lines for the next few instructions.

This software cache control mechanism can be considered to be one of the biggest differences between the MAST model and current cache management schemes. Since software cache control is much more flexible than some fixed, program-independent policies, it provides a good implementation environment for new cache management schemes.

5.2.2 Program Understanding

The consequence of "pure hardware-managed cache control" in current cache policies is that cache is transparent to the system software and program understanding in cache management is considered as impossible. In other word, information about future references is not available to any cache management schemes[2]. As a result, most (if not all) cache management schemes are either based on history of execution or a probabilistic model (e.g. Markov Chain model [Spi76] [Spi77]). The MIN algorithm, which was proposed by Belady in 1966 for straight line code [Bel66], is always considered as an "unrealistic optimal" replacement

[2] Even in cache prefetch, *most* prefetch schemes only fetch line $i+1$ when there is a cache miss for referencing line i.

algorithm.

On the other hand, program understanding and knowledge about future references is not difficult to obtain and can be implemented in the MAST model. This is especially true with current compiler and program flow analysis techniques. In fact, this program understanding phase in the MAST model is "almost free" because it is the basic step for most compiler optimization techniques. Although program flow analysis might not provide an exact reference pattern of a reference program, it will limit the number of possible choices to a small number. Moreover, the exact control flow and the reference pattern within a basic block is very easy to obtain. This capability of program understanding in the MAST model provides a large cache performance improvement over current cache management schemes because this information can be used to determine when and how a cache line should be placed in cache and/or replaced from cache. Cache pollution can be eliminated because information is fetched from main memory only if it is needed by a reference program.

5.2.3 Order of Referencing

Due to the lack of program understanding in current cache management schemes, the order of reference of instructions and data in a reference program is completely ignored — only the history of execution or probabilities are used in current cache control schemes. This causes a large performance loss because cache line replacement cannot be based on the usefulness of information currently in cache to future references.

For example, when a line γ in a program is referenced, the principle of locality of references suggests that the probability of referencing line γ again in the near future is very high. However, if that reference to line γ is the last reference to that value (i.e. the last use in its D-U chain), line γ will be of no use to the rest of the program.

In some cache management schemes, a runtime profile is sometimes used in cache prefetch. However, this partial order of references is only good to guess what is going to be used. It does not indicate the relative usefulness of information in cache in future references.

In the MAST model, the complete order of references is used in the analysis of machine level state transitions. This provides more accurate information about when and what lines in cache should be replaced, hence,

the cache performance using the MAST model is higher.

5.2.4 Cache Utilization

The primary function of cache is to bridge the speed gap between the CPU and the main memory by keeping copies of currently referenced data in cache. Hence, the fundamental operational assumption of any current cache management schemes is *to place as many references in cache as possible so that more information can be referenced from cache.* In other words, maximizing cache utilization is assumed to maximize system performance. Consequently, the cache hit ratio is the most important measuring parameter for cache performance and all cache management schemes are "cache-through" — when there is a cache miss, information needs to be placed in cache before (or simultaneously) it is referenced.

While cache utilization might sometimes indicate how well the system performs, it is *not always true* that maximizing cache utilization maximizes system performance. There are at least two reasons for this argument:

- In a computer system, the main bottleneck of the system performance might not be the cache utilization. For example, with the development in supercomputing and VLSI, parameters like total memory access time or bus traffic of a system are far more important than the cache hit ratio.
- There are many situations in a reference program where the system performance is actually *decreased* if it is placed in cache. When a line is placed in cache, there are overheads associated with the line placement and the possibility of replacing "useful" lines already in cache. Unless the benefits of having that line in cache is greater than its overhead, it is better to just reference the information directly from main memory. In order word, only those information with high reference frequencies should be placed in cache and the rest should be referenced directly from main memory.

From our simulations (discussed in Chapter 7), it is shown that the relationship between cache hit ratio and total memory access time is unclear. For example, a very high cache hit ratio might correspond to large memory access time. This is another significant difference between the MAST model and current cache management schemes. In the MAST model, total memory access time is the measuring parameter while cache hit ratio is

used as the measuring parameter in current schemes.

5.2.5 Cost Consideration

In current cache designs, since cache hit ratio is the primary measuring parameter and program understanding is not part of the management schemes, the relative costs for each cache operation cannot be modeled accurately. The cost of replacing a "dead" cache line and that of replacing a "live dirty" cache line are always assumed to the same. Moreover, for a given cache size, the effect of increasing line size to the cost of cache line replacement is often ignored.

This inaccurate modeling of the cost of cache operations cause a large performance loss. As is discussed earlier in last section, cache operation decisions should be based on the relative cost and benefit of placing a line in cache, which is both architectural and program dependent. With different relative costs and benefits for placing a line in cache, operation decisions (e.g. cache bypass or cache through) might be different. In VLSI processors and in multiprocessor environment, where costs of cache operations might differ by a factor of 10 or 100, this cost consideration becomes more important in cache management. Almost all these are ignored by current cache design.

In the MAST model, the decision of "cache-through" and "cache-bypass" for references and cache line replacement is based on the relative cost and benefit of placing a line in cache. Hence, a large fraction of the performance loss due to inaccurate cost modeling is regained.

5.2.6 Reference Liveness

In current cache management schemes, references are based on names or physical memory addresses. Due to the lack of program understanding in cache management schemes, information in cache is always considered as valid and useful to future program references. However, this is not always true. When a *value* stored in cache is not referenced in the rest of a reference program, it can be considered as "useless or dead" and the cache cell storing this value can be considered as *empty*. This happens if:

- the last reference of the memory address of a value has been made, or

- the next reference of the memory address of a value is a memory write.

In other word, the concept of live range of references in register allocation [AhS86] [Die87] is completely missing in current cache designs. Invalid information might be kept in cache while valid information in cache is replaced due to insufficient cache space.

The principle of locality of references is insufficient to describe the "def-use chain" of references in a program. This is because after the last use of a value in a program, the concept of live range of references would mark this value as dead and this value should be the next one to be replaced. However, the concept of locality of reference suggests this value should be kept in cache.

In the MAST model, the liveness of a value is accurately defined (as is described in Chapter 3) and is used in the cache control. Hence, any unnecessary storage and update of "dead" values in cache is avoided and the performance loss due to dead values in cache is regained.

5.2.7 Summary

In the last six sections (from Section 5.2.1 to Section 5.2.6), fundamental differences between the MAST model and current cache management schemes were discussed. It is clear that there are some concepts about program reference behavior (e.g. reference liveness) and cache operations (e.g. cost consideration) missing in current cache designs. Consequently, there is a certain fraction of cache performance which is very difficult (if possible) to obtain with current cache management schemes.

The rest of this chapter is devoted to the discussion of how current cache management schemes should be modified so as to include some of these concepts. In particular, we would like to study improving cache performance with:

- live range analysis only,
- cache bypass only, and
- software cache control without cache bypass.

As is stated at the beginning of this chapter, the main goal of this discussion is to develop some simple linear runtime algorithms which can regain major fractions of cache performance loss in current designs.

5.3 Improving Cache Performance with Live Range Analysis

The definitions of live range of a value and an instruction in cache were given in Section 3.4.1 and are repeated here:

Definition 3.5: Live Range of a Value

> The **live range of a value** λ is defined as the set of instructions during which the value λ exists and may be referenced. In other words, it is the *D-U chain* of $\lambda \cup$ all instructions which, on some flow path, may be executed after λ's *def* and before the last *use* of λ on that flow path.

Definition 3.6: Live Range of an Instruction

> The **live range of an instruction** κ is defined as the set of instructions, including κ, which may be executed after the first execution of κ and before the last execution of κ on some flow path. Notice that for straight-line code the live range of an instruction κ is always the set $\{\kappa\}$, however, if κ is enclosed in a loop or multiple-caller subprogram the set may be greatly enlarged.

Note that the live range of a value or an instruction is defined in terms of *values* instead of names or labels.

The live range of references in a program is very important to cache management schemes. With the live range defined for each reference value, the longest period (or the largest segment of code) that a value should possibly be in cache can be determined. Furthermore, since the number of simultaneous live values at each cache stage in a reference program can be found, this gives an upper bound of the cache size needed for a reference program: *the upper bound of the cache size needed for a reference program is the maximum number of simultaneous live values at any cache stage in a reference program.* Once the cache size is beyond this limit, no further performance improvement should possibly be made if cache control is managed correctly. This is because the cache is large enough to hold *all* live information at any cache stage.

5.3.1 Previous Research on Live Ranges of References

Before investigating the mechanism for, and benefits of, live range analysis in current cache management schemes, it is useful to briefly survey characteristics of live ranges in programs. Although the concept of live range of references does not exist in cache management schemes, this survey is still possible because live range of values is well defined in register allocation and program flow analysis [AhS86] [Cho83] [ChH84].

Some of the characteristics of live ranges of values in a program and its implications to cache designs are summerized as follows:

- *Live Values vs Program Size*

  The average number of simultaneously live values at any execution point of a program is only slightly dependent (if not independent) on program size. Although it is generally true that larger programs tend to have larger average number of simultaneously live values, the increasing rate of this number with program size is extremely slow (or almost zero). Moreover, an incremental increase in this number usually corresponds to a large increase in the program complexity. This implies that cache size selection for a machine should be independent on the size of application programs running on it.

- *Number of Simultaneous Live Values*

  The average number of simultaneous live values (local and global) at any execution point in a program is usually relatively small, usually from 50 to 100. This implies that the theoretrical upper bound for cache size needed should be relative small if cache is managed properly and intelligently. However, in current cache design, cache size of 4Kbytes to 16 Kbytes is quite common and cache size less than 1Kbyte is considered as small. The main reason for this is that current cache design is managed dynamically by hardware and a major fraction of the cache area is always under-utilized. To support this argument, it is shown in Chapter 7 that the functional difference between cache and registers is the ability to handle ambiguous references. If register file with size 32 or 64 is considered large enough for most applications and the ratio of ambiguous references to unambiguous references is about 1:1 to 1:3, the cache size needed should not be very large (e.g. 16Kbytes).

- *Length of Live Range of References*

  From the study of live range of references of programs by Lunde

[Lun77], the statistic for the distribution of the length of live range with respect to the percentage of total program references is summarized in Table 5.1:

Table 5.1: Distribution of the Length of Live Ranges of References

Length of Live range (in words)	Cumulative % of total references
1 - 1	0.09
2 - 3	0.48
4 - 7	0.77
8 - 15	0.90
16 - 31	0.96
32 - 63	0.98
64 - 127	0.99
> 128	1.00

This table shows that about half of the references have live range of length 1 to 3. Over 90 percent of the references have live range of length less than 16. This implies that the time for most references to be kept in cache should be quite short. However, once a datum is dead after the reference, it will be kept for a period of time before it is replaced. For example, if LRU is used as the replacement policy, a dead reference will remain in cache for a period at least greter than the cache set size unless the reference distance between the last use and the def of a memory location is short. Consequently, for cache with a high set associativity, the live period of a value in cache is much less than its dead period.

This shows why live range analysis is so important to current cache management schemes and how cache performance can be obtained with this analysis. In the next few sections, modification of current cache management schemes such as LRU, FIFO, random and Belady's MIN

algorithm to include live range analysis are given.

5.4.2 LRU With Live Range Analysis

The least recently-used scheme (LRU) for cache replacement chooses for replacement that line in cache which has not been referenced for the longest period of time [Spi76]. It is organized as a LRU stack, which is a list of all cache lines referenced by a program in order of most recent use. When a line is referenced, it is placed on top of the LRU stack. If the referenced line is in cache, any lines above it in the stack is pushed down by one slot. If the referenced line is not in cache, all lines in the stack are pushed down by one slot and the bottom line in the stack is removed.

Suppose the set size of the cache (i.e. the size of the LRU stack) is κ. Once a line α is referenced from cache, three possible situations might happen afterward:

- If this is the last reference of line α, it takes κ distinct line references to replace line α from the cache.
- If line α is going to be referenced after λ other distinct line references, two possible cases might occur:
  - $\lambda > \kappa$

    Line α is continually pushed down in the cache until the κ^{th} distinct line reference (after line α is referenced) is made, when it is replaced from the cache.
  - $\lambda \leq \kappa$

    Line α is pushed down the LRU stack until it is referenced again.

In all these cases, line α is kept in cache, but is not referenced for a period of time.

Now suppose that line α is dead immediately after it is referenced. A "hole" — a cache cell that contains invalid or useless data — is created in the LRU stack and this hole will remain in the stack for the period as described above. This certainly decreases system performance because a hole in the LRU stack implies one less cache cell for storing information.

To analyze the impact of these "holes" in the LRU stack compared to the effective size of a cache, suppose that a line α is in cache and that the processor is about to make what is known to be the last reference to α. In either LRU or an LRU approximation, the line α would be present in cache

for $O(\kappa)$ time units after the last reference, where κ is the number of lines in a *cache associative set*, because it will take that long for κ to be nudged into the least-recently-referenced position. In effect, if the average cacheable item is referenced τ times, then approximately $1/\tau$ of the cache cells will be wasted. Notice that τ=1 items would be a complete waste — something referenced only once should never be placed in cache. Further, note that in typical programs, relatively few items are referenced more than a few times (except perhaps in some loops).

With the information of "liveness" of references available to cache management schemes, these "holes" in the LRU stack can easily be eliminated. Suppose a line α in cache is dead immediately after it is referenced. All lines below line α in the LRU stack can be popped up by one slot to fill the hole. Lines above line α in the LRU stack remain unchanged in their LRU stack positions during this process. If α is not in cache, the reference should be made directly from main memory, bypassing the cache. Even if the cache bypass option is not available, some slot in cache (e.g. the bottom line of the LRU stack) might be used to temporarily hold line α. After the reference to line α is finished, the hole created can be removed using the techniques just described.

More specifically, the modified LRU cache management scheme with live range analysis is as follows:

(1) The live range of all references in a program are determined.

(2) For each reference line α, if line α is live after the reference, the current LRU management scheme is performed. Otherwise, line α is dead after the reference is made and there are two cases:

   (a) line α in cache

   After the reference to line α is made, memory update of line α is performed if necessary (i.e. if line α is dirty and the liveness of line α is not precisely known). Then all lines below line α in the LRU stack is popped up by one cache slot.

   (b) line α not in cache

      (i) If cache bypass option is available, the reference is made directly through main memory, leaving the cache content unchanged.

      (ii) If this cache bypass option is not allowed, memory update of the bottom line in the LRU stack is performed if it is dirty.

Then line α is placed at the bottom slot of the LRU stack and is referenced. Line α is marked as empty or invalid after the reference is made.

5.4.3 Other Schemes With Live Range Analysis

Extension of other cache management schemes such as FIFO, random replacement [Smi82] and Belady's MIN algorithm [Bel66] to incorporate the live range concept of references is very similar to that described in the last section for LRU replacement scheme and is not repeated here. The basic idea is to detect the occurrence of a "dead" cache line and to mark it as "invalid". Rearrangement of the queue in cache to remove the "hole" due to dead cache line is performed (if necessary).

In FIFO, the techniques used in the LRU scheme with live range analysis can be directly to the FIFO to eliminate dead cache line. The only difference is that when cache line replacement occurs, the line to be replaced is the one residing in cache for the longest period of time, instead of the least recently used line.

In random replacement, a dead cache line can be marked as "empty" and no cache line rearrangement is necessary. In Belady's MIN algorithm, it should be modified as: invalid (or dead) cache line should be chosen for cache line replacement; and if all cache lines are live, the cache line which will be referenced furthest from the current stage is selected.

When the cache line size is greater than one, a situation where some elements in the line are live while others are dead might occur. A simple solution to this problem is to mark a cache line live if there exists at least one live element in that cache line. Note that live range analysis has greater impact to cache design with larger cache set size and/or smaller cache line size. With larger cache set size, the holes will remain in cache for a longer time before it is replaced. With smaller cache line size, the problem of partial liveness of a cache line is reduced.

5.4 Improving Cache Performance by Selective Cache Bypass

Since cache reference time is so much less than main memory reference time, it is commonly held that as many data as possible should be placed in cache and one typical measure of the efficiency of a cache design is the cache hit ratio. The problem is simply that it is not always beneficial to fetch a line into the cache on a cache-miss even if the cache is infinitely large — *increasing cache hit ratio sometimes reduces system performance!* Other criteria like memory traffic have occasionally been used instead of cache hit ratio, but these measures are also somewhat imprecise and indirect. If one wants to *minimize total memory reference time*, then that is the obvious measure by which cache performance should be judged. Throughout this thesis, cache performance is measured in terms of the effect on total memory reference time.

Why are the more commonly used cache performance criteria inaccurate measures of system performance? One reason is that there is always an overhead associated with fetching a line from memory into cache. If the benefit gained from having that line in cache is not greater than the overhead that loading the cache line implies, then it is faster to reference the data of that line directly from main memory. This is true even if the cache is infinitely large, but is even more dramatically true with smaller caches. If some mechanism can be used to selectively disable or bypass the cache for those references which cache cannot improve:

- the cost of loading the cache with these lines is saved and
- for finite-size caches, more cache space becomes available to other references and the probability of accidentally replacing useful lines (those lines that can help improve system performance) is reduced, i.e., there will be less cache pollution.

Simulation results, reported in Chapter 6, strongly support this view. An *average of 10% to 30% reduction in total reference time* can be achieved simply by using this proposed cache bypass mechanism.

5.4.1 Current Cache Designs and Bypass Concepts

Before investigating the mechanism for, and benefits of, selective cache bypass, it is useful to use an example to illustrate the importance of cache bypass in current cache management schemes. In part, this highlights where the extra performance comes from, but it also clarifies the constraints these current policies impose on the cache bypass mechanism. It illustrates why some constraints imposed by current cache replacement policies often cause a large decrease in system performance, as well as how eliminating some of these constraints can regain much of the lost performance.

This discussion also briefly describes the cache bypass mechanism used in the C1 minisupercomputer manufactured by Convex Computer Corporation [Con86]. Although the strategy used for cache bypass in the C1 is very limited, it does demonstrate the importance of incorporating a bypass mechanism.

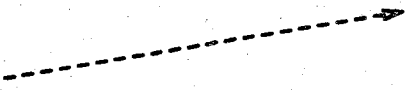
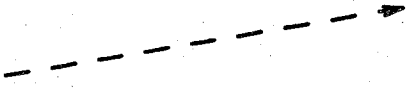
5.4.1.1 Example of Cache Bypass

Each of the current cache replacement policies has its own unique technique for placing and/or replacing cache lines. However, the option of deciding not to put the requested line in cache was not considered. In all conventional cache replacement policies, immediately after each reference, the line referenced must be placed in cache. This implies that whenever a cache miss occurs, the missed line needs to be fetched into the cache and this line fetch is independent of whether the fetched line would bring improvement to system performance.

The main argument for this constraint is that since reference time of data in cache is much smaller than that from main memory and with spatial and temporal behavior of program references [Spi77], having the current referenced line in cache has a high probability bringing improvement in system performance. While this argument is generally true, it is possible to predict with some certainty which lines will not contribute to improving performance; without such prediction, it is easy to envision scenarios where the cache would replace lines it should have kept with lines that will never again be referenced. This leads to a worst-case scenario in which a machine runs slower with cache than without it. Bypassing the cache, hence avoiding this pollution, this worst-case scenario is averted.

An example of this problem is easily found. Suppose there is a fully-associative cache of size two, line size one, and the memory reference string is *123123*. (It is interesting to note that this example is exactly the kind of reference sequence one would get in executing a typical loop which references more data than there are cache cells — which is the worst-case for LRU.) With the cost of different types of memory references shown in Table 5.2 (and the line-style used to represent each), the cache content after each reference with random replacement, LRU, and modified LRU with cache bypass mechanism are shown in Figures 5.1, 5.2, and 5.3.

Table 5.2: Cost for Each Type of Memory Reference

Line Pattern	Cost (Time)	Type of Reference
	*none*	—
	T_c	Reference from Cache
	T_r	Reference from Main Memory
	$T_c + T_p$	Reference through Cache (with Fetch to Empty Cache Line)
	$T_c + 2(T_p)$	Reference through Cache (with Replacement of a Cache Line)

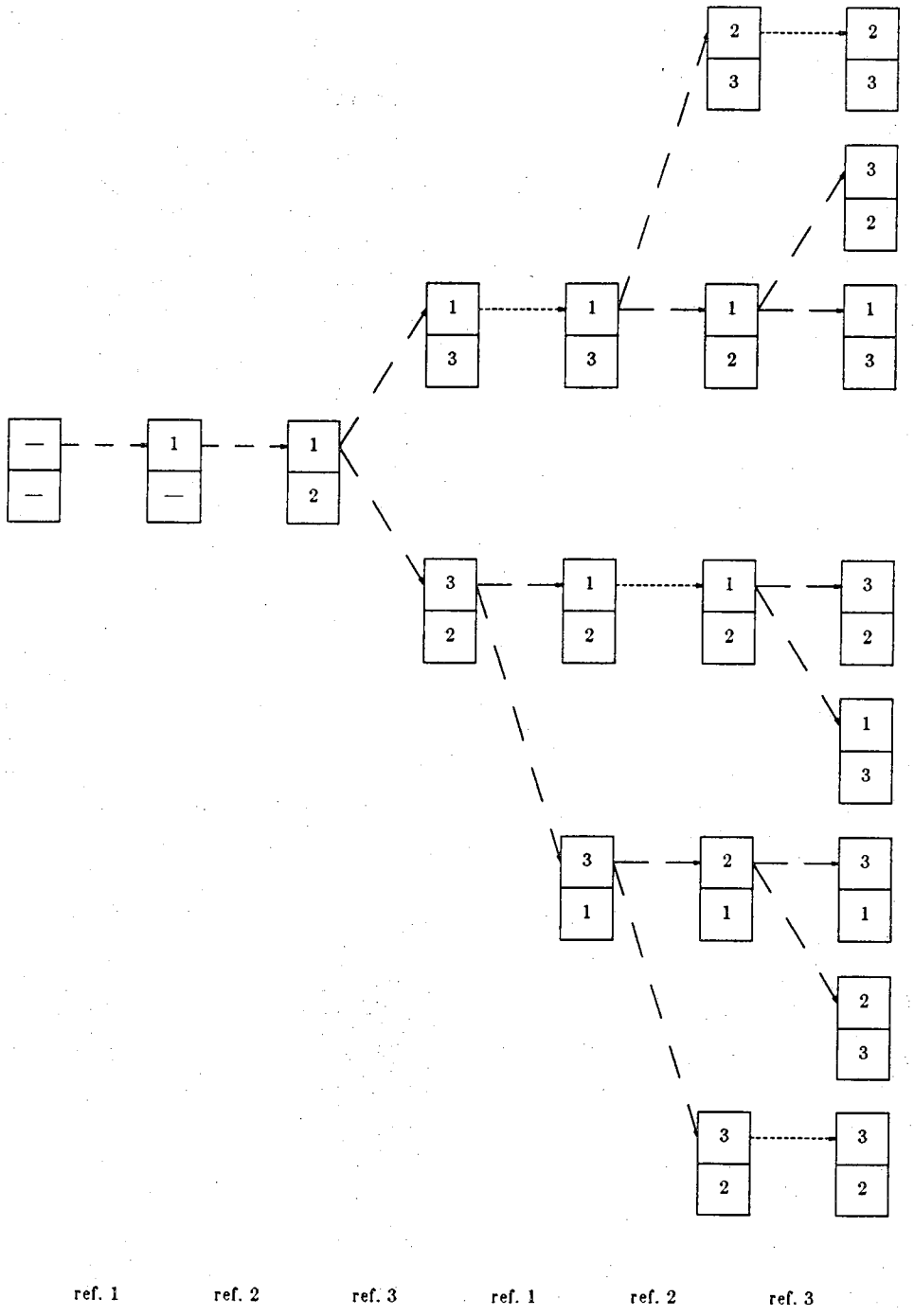


Figure 5.1: Random Replacement Transactions for *123123*

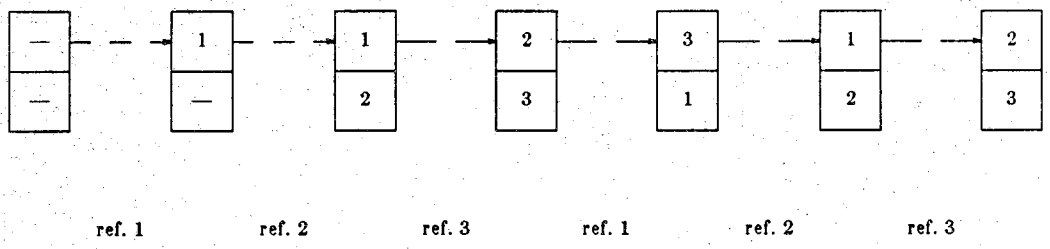


Figure 5.2: LRU Transactions for *123123*

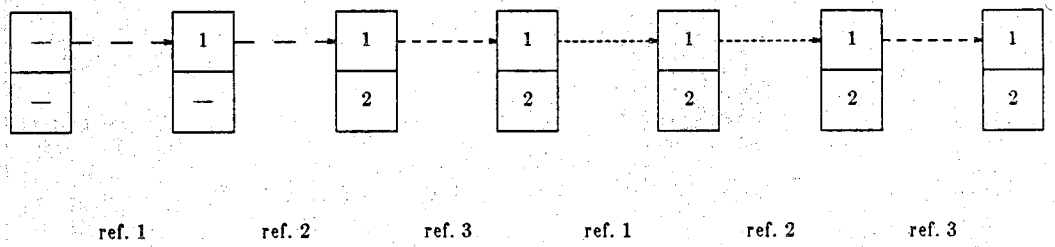


Figure 5.3: Modified LRU with Cache Bypass for *123123*

Table 5.3: Comparison of Execution Times for *123123*

Cache Policy	Cost	Cost with $T_p = T_r = 10T_c$	$Cost_{Cache\text{-}Policy}/Cost_{Bypass}$
Bypass	$2T_p + 2T_r + 4T_c$	$44T_c$	1.000
Random	$7.75T_p + 6T_c$	$83.5T_c$	1.898
LRU	$10T_p + 6T_c$	$106T_c$	2.409
No Cache	$6T_r$	$60T_c$	1.367

The total reference costs using these three policies are shown in Table 5.3. In this table, it can be seen that the ratio of $Cost_{Random}/Cost_{Bypass}$ is 1.898 and the ratio of $Cost_{LRU}/Cost_{Bypass}$ is 2.409.

Notice that while placing data *1* and *2* in cache can improve system performance, placing datum *3* in cache actually decreases the system performance. Unfortunately, if cache bypass is not considered, the resulting performance is the worst possible — in fact, it is worse than if no cache were present. With selective cache bypass, one might simply reference datum *3* directly from main memory; yet the cache would speed-up references to data *1* and *2*.

5.4.1.2 History of Cache Bypass

Although not commonly accepted as part of current cache design, cache bypass is not entirely new.

Nearly all cache-based computers have some provision for disabling the cache so that memory-mapped I/O transactions can take place. However, the idea of enabling/disabling the cache for each memory reference is not well supported by most of these systems (presumably the possibility had not been considered). These systems typically require an entire instruction to be executed to change the cache enable state. Despite this, such systems can be used to implement cache bypass where several consecutive references should be bypassed.

Some machine designers also recognized that the performance of cache could be improved by simultaneously requesting each datum from both main memory and cache. In this scheme, if the item is found in the cache then the cached value is used and the main memory request is cancelled or ignored. If not, the item is returned directly from main memory to the processor, simultaneously initiating a cache update for that datum's line. This technique does improve performance, but may require fairly expensive hardware and does not avert cache pollution — it merely reduces the cost of referencing "through" the cache.

Somewhat closer in spirit to our approach, Convex Computer Corporation has implemented a selective cache bypass mechanism in their C1 minisupercomputer. The strategy employed is [Con86]:

> Upon load or store, the physical control unit either writes the referenced data into its cache or bypasses the cache and accesses

> main memory directly, leaving the cache unmodified. All aligned 64-bit vector loads and stores result in cache bypass. Loads and stores of aligned, contiguous 32-bit vector elements bypass the cache as well. Since vector accesses dominate supercomputer-class applications software, cache bypass opportunities occur frequently.

Apparently, the cache bypass mechanism is employed only on vector operations because the C1 has a cache with a set size of one, hence, loading a vector register had the effect of totally flushing the cache — obviously negating any benefits of caching. In any case, the Convex scheme is quite reasonable, and was sufficiently new so as to be patented (patent pending?); the problem is that it equates "vector" with "bypass," and this isn't really correct. Some vectors *should* be cached and some scalars shouldn't be, but on the average the Convex scheme is right often enough to yield a big improvement.

In contrast, the current proposal for cache bypass in this thesis is to use a compile-time static analysis of the reference behavior of each program to compute a "cache/bypass" tag for each memory reference the compiled code makes. These tags are used at runtime to control a cache enable/disable line.

5.4.2 Implementing Cache Bypass

As shown in the example of Section 5.4.1.1, LRU referencing of all data through the cache actually performed worse than if no cache were present.

There are two main reasons for this phenomena. First, there is often a large time overhead implied in moving lines of data between cache and main memory. This overhead increases as the cache line size is increased. Consequently, fetching a line into cache can improve system performance *iff* the total number of references to data in that line (before that line is replaced) is such that the savings in referencing cache outweighs the overhead of moving that line between cache and main memory. If not, the total time to make these references will be minimized by ignoring the cache — bypassing to directly reference main memory. Even if the cache is infinitely large, this still holds.

Second, since all real caches are finite, placing one line in cache generally means that some other line cannot be in cache. Hence, placing infrequently referenced lines into cache not only adds a large overhead to

total memory access time, but also prevents speed-up that could have been gained if some other (more heavily referenced) line were placed in cache. This effect is what we call "cache pollution."

Since minimizing the total memory access time is our goal in selective cache bypass and the total access time depends on both the architectural design and the implementation technology of the cache and main memory, some details must be supplied. In the rest of this section, we have chosen to discuss cache bypass assuming that the supplied information is that of a typical system.

In Section 5.4.2.1, a brief discussion of current integrated circuit technologies and their impact on memory access time is given. Criteria or rules to determine whether a reference request is going to bypass the cache and to reference directly from main memory are presented in Section 5.4.2.2. Section 5.4.2.3 gives a very simple and cheap, yet efficient, way to incorporate a cache bypass mechanism with an LRU policy. Other cache management schemes with cache bypass mechanism are discussed in Section 5.4.2.4.

5.4.2.1 Integrated Circuit Technologies

Integrated circuit (IC) technology is one of the major parameters in the design of a cache bypass mechanism (discussed in the next section). Hence, a brief survey of current different (IC) technologies and its impact on off-chip and on-chip memory reference time is necessary. Table 5.4 gives the on-chip and off-chip memory access times for some of the current integrated circuit technologies [MiF86]. From this table, we see that the ratio of off-chip to on-chip memory access times is at least 10. Using this ratio, an estimate of the minimum reference frequency that a line needs to justify its placement in cache can be obtained.

Table 5.4. Memory Access Time of Different IC Technologies

Type of Access	Silicon CMOS/SOS	Silicon NMOS	GaAS
On-chip memory access	10-20ns	10-20ns	0.5-2.0ns
Off-chip on-package memory access	40-80ns	20-40ns	4-10ns
Off-chip off-package memory access	100-200ns	100-200ns	20-80ns
Ratio of off-chip on-package to on-chip memory access	4	2	5-8
Ratio of off-chip off-package to on-chip memory access	10	10	40

5.4.2.2 Criteria for Cache Bypass Mechanism

Throughout the current work, the main focus is the reduction of total memory reference time for a program. Hence, criteria proposed here are based on the comparison between the time overhead involved in having a line in cache and the total reference time saved by referencing data in a line in cache.

The time overhead of placing a line in cache is the transfer time for all data of that line from main memory to cache. If any dirty[3] line is bumped out of cache using a write-back cache, a similar transfer time to update the main memory is also included in this overhead. Since the amount of data transfer between main memory and cache is constant for a cache design, this overhead is only architecture design and implementation technology dependent, and is independent of program behavior.

On the other hand, the time savings for placing a line in cache accumulates every time data in that line is referenced. Hence, the savings are, in addition, program dependent.

---

[3] A line in cache is considered dirty *iff* some portion of the value it contains does not match the value stored in the corresponding main memory line.

There are additional factors which can influence the costs and the savings of placing/replacing a line in cache, resulting in slightly different cache bypass decisions for references in a program. For example, if a reference is going to bypass the cache and directly access main memory, the average probability of bumping a line from cache decreases, and cache space could also be viewed as available to other lines.

To define an algorithm for determining when to bypass the cache for a particular reference, some definitions and notations are useful.

$overhead(\lambda)$ = time overhead of placing/replacing line λ in cache.

$saving(\lambda)$ = time saving of having line λ in cache before it is replaced.

$\eta(\lambda)$ = reference frequency of line λ in cache before it is replaced

$Time_{cache}$ = access time of a datum from cache

$Time_{main}$ = access time of a datum from main memory

$Time_{place}$ = time taken to transfer a line from main memory to cache

$Time_{replace}$ = time taken to update a cache line and to transfer a line from main memory to cache

With the above notations, $overhead(\lambda)$ and $saving(\lambda)$ are calculated as follows:

If no dirty line is bumped out of cache, the overhead is:

$$overhead(\lambda) = Time_{place}$$

If a dirty line is replaced (bumped) from the cache, then the overhead is:

$$overhead(\lambda) = Time_{replace}$$

The savings for having line λ in cache (before it is replaced) is:

$$saving(\lambda) = \eta(\lambda) * (Time_{main} - Time_{cache})$$

In order for a reference line λ to bypass the cache, the overhead $overhead(\lambda)$ must be greater or equal to the total time savings $saving(\lambda)$. Only in this case can the placement of line λ contribute to improve system performance.

5.4.2.3 Algorithm for LRU Cache Bypass

In this section, LRU cache replacement is used as the basic scheme and the cache bypass control is added on top of this policy. Since LRU policy is probably the most commonly used and most commonly expected to yield good performance, the comparisons of simulated performance with/without cache bypass (in Chapter 6) are very good estimates of the expected improvement derived from converting commonly available computers to use a cache bypass mechanism.

A fast, simple, efficient (yet sub-optimal) algorithm to determine when a reference should bypass the cache is proposed here. The algorithm is based on the concept of a trace, as discussed in trace scheduling techniques used for automatic parallelizing compilers [Ell85]. The procedure to determine, for each reference in the program, whether to bypass or to reference through the cache is:

(1) Perform traditional flow analysis and build the program flow graph. (This step should be considered "free" because any *good* compiler will use this same analysis to aid in generating efficient code.)

(2) For each trace (a possible control flow path which has not yet been processed), do the following:

   (a) Mark all references in this trace as "cachable" (put in cache).

   (b) Scan this trace, keeping track of which items would be resident in cache assuming that all items marked as cachable are always referenced through the cache and that LRU is used to determine which item is bumped from cache when line replacement occurs. As the references are scanned, the time overhead and savings realized for each cachable line are accumulated. As a simple heuristic, the savings for referencing an item within a loop is multiplied by a factor[4] of 10.

   (c) At the end of the trace, mark all references which have a larger overhead than savings as "non-cachable".

---

[4] This is a rough approximation to weighing each reference in the trace by its expected number of executions — it assumes each loop executes an average of 10 times. If the compiler has a better estimate, this can be used instead. Techniques for the compiler to make more intelligent estimates of expected execution frequencies are discussed in [Die87].

(d) The above set of markings can be somewhat improved, although not made optimal, by repeating steps 2b and 2c. Such repetition is, however, completely optional. All the simulation results given in Chapter 6 used only a single pass.

This algorithm, although very crude and simple, reaps speedups ranging from a few percent to a factor of nearly 100, depending on the cache configuration and the benchmark used. Speedups greater than 2 are not unusual for commonly used cache configurations.

5.4.2.4 Other Cache Replacement Schemes with Cache Bypass

Algorithms to determine when the cache should be bypassed for each reference in a program using other cache replacement schemes is very similar to that in LRU discussed in the last sections. The only difference is to use other replacement schemes to determine the replacement of cache lines, instead of using LRU in step 2(b) of the algorithm.

For example, Belady's MIN algorithm with cache bypass mechanism should be modified as follows: cache line placement only occurs for those references which are not bypassed; if cache line replacement is necessary, the cache line which is referenced furthest away from the current referencing stage is selected. The hardware required for any replacement management scheme is still the same — one bit per reference.

The cache bypass idea works best when the replacement policy is deterministic under a given cache configuration — at any stage in a reference program, the cache line to be replaced is precisely defined for a given cache configuration. Examples of this are LRU, and FIFO. For random replacement, the decision of when to bypass the cache is harder to make because the savings from placing a line in cache might not be precisely known (due to its probabilistic nature)[5].

[5] This might not be true if the word "random" just means that the replacement policy is based on some complex function. In this case, random replacement is not truly random and the approaches suggested for "deterministic schemes" can be applied.

5.5 Hardware Implementation of Live and Bypass Control

Once the liveness and bypass control information of references in a reference program are found, they can be transmitted to hardware control logic using techniques discussed in Section 4.9 for the MAST model. For a reference liveness bit, a "1" means "live" and a "0" means "dead". Similarly, a "1" in a cache bypass bit means "bypass" and a "0" means "go through the cache".

Comparing the live range or cache bypass model to the MAST model, the number of cache control bits per reference required is as follows:

(1) MAST model

The total number of cache control bits required is *log(set_size)* + 2. The *log(set_size)* bits is used to decide which cache line should be replaced, one for cache bypass, and the other for the liveness of a datum after it is referenced (whether replacement of a dirty line needs any memory update).

(2) Live-Cache and/or Bypass-Cache model

The total number of cache control bits required is two — one for cache bypass, the other for the liveness of a datum after referenced. Note that this hardware requirement is true for any cache management schemes with some fixed replacement rules (e.g. LRU, FIFO).

Since the liveness and cache bypass bits are independent of one another, each can be implemented without the others.

The natural question is: how does the compiler get these two bits of information for each reference into the cache control at runtime? There are a number of alternative solutions to this problem and each of these solutions trades off some resources or capabilities.

The conceptually easiest and most efficient way to transmit this cache control information is to embed two bits in each instruction for each memory reference the instruction may cause. For a new machine design, this is fairly convenient; reserving two control bits to obtain speedups of total memory access time by factors of 2 or more is virtually always worthwhile. Also, existing machines with at least two currently unused bits in each instruction should probably use this implementation. An good example is the Precision Architecture from HP. In the original architecture design, two bits are reserved for "cache hint" [Hol88]. Currently, these bits are still free to use, even though the machine design was finished a year ago. For machines with

only one unreserved bit, the cache-bypass control bit is recommended because performance improvement gained by the cache bypass bit is usually greater than that by the reference liveness bit.

Alternatively, the instruction set of the machine can be expanded to include explicit cache liveness and cache bypass control instructions. In fact, cache bypass instructions exist for virtually all computers which have cache. An extreme example of this explicit cache control is the IBM 801 [Rad83], where individual cache lines can be explicitly allocated and deallocated; most systems simply permit the cache to be enabled/disabled as a whole. Since bypasses may come in "clumps", even this crude bypass control can gain some improvement; however, bypasses do not always come in clumps.

By defining a new instruction specifically to implement reference liveness and cache bypass controls, one could permit each cache control instruction to set the pattern of liveness and bypass decisions for the next λ references, where $2 * \lambda$ bits is somewhat less than the machine word length. Again, some performance would be gained, but the high frequency of these explicit cache control instructions might limit its performance.

While all the above schemes have some merit, there is another scheme which both permits one or two cache control bits (either reference liveness or cache bypass control) to be associated with each instruction and does not require changes in the instruction set design or encoding. In current machine designs, the addressable space is typically very large and programs rarely use the entire addressable space of the machine. Thus, it is possible to trade one or two address bits (e.g., the most significant bits of an address) for use as the control bits for the reference liveness and/or cache bypass. In fact, this solution is suggested by Intel in their 80386 programmer's reference manual [Int86] as a way to provide a cache control bit for use in multiprocessor cache coherency control. Worst case, this effectively reduces the addressable space by 50% (for one cache control bit) or 75% (for two control bits)[6]. Of course, it also causes the compiler writer a bit of grief in that not only must all addresses be correctly tagged, but the compiler must also be careful about operations such as pointer arithmetic or comparisons.

---

[6] The actual address space may not be affected because address mapping mechanisms may be able to circumvent the loss.

Other methods, such as using a separate cache controller to explicitly control the cache (similar to the remote PC idea [Rad83] and the program skeleton idea [Bre87]) are also possible. However, the overhead and the synchronization cost involved may be too large to be practical.

5.6 Software Cache Replacement Schemes

A proposed software cache replacement control scheme without cache bypass and reference liveness analysis is a simplified version of the MAST model where only the choice of cache line replacement is determined explicitly by software. Any reference has to go through the cache (i.e. no cache bypass option) and the liveness of datum after it is referenced is unknown. In the hardware implementation, only *log(set-size)* cache control bits are needed for the selection of cache line replacement.

The algorithm of this software cache replacement control scheme is the same as the one proposed in Chapter 3 and 4, except that any cache bypass option is unavailable. That is, any transition edge, which connects two successive cache states with the same cache configuration with the transition reference not in cache is eliminated from the MAST graph. For each cache stage, there is one cache state less and for each cache state where replacement occurs, there is one edge less coming from that state.

The computational complexity of this scheme is lower than that of the MAST model, but is higher than that with reference liveness and cache bypass analysis. The hardware complexity of this model is also in between of the MAST model and the cache liveness/bypass models. The only exception is when the cache set size is one. In this case, no cache control bit is required because there is no choice for cache line replacement and the scheme becomes the current direct-mapped cache model.

In this model, since cache line replacement can be explicitly defined in cache control bits, a large fraction of the unnecessary cache line replacement in current cache management schemes can be avoided. For example, when the loop size is greater than the cache size, any cache line replacement would be forced to use the last line of each cache set, leaving the other cache lines unchanged (i.e., achieving similar goal as the cache bypass).

Due to the lack of cache bypass option, lines need to be fetched into the cache even if the overhead of fetching the line is greater than the saving of placing it in cache. Also, unnecessary main memory update might occurs

due to the lack of reference liveness bit.

Since this model explicitly chooses a cache line to be replaced, it is expected that it has larger effect in cache with larger cache set size than in cache with a smaller cache set size. With small cache set size, the possible choice of line replacement is very limited. Hence, the performance difference between the software cache replacement model and current cache management schemes (e.g. LRU) is very small. In fact, when the set size is one, they are identical.

5.7 Conclusions

In this chapter, applications of the MAST model to current cache structures are discussed. By comparing the MAST model with current cache design models, it is found that there are cache operational constraints in current cache designs which severely limit its performance. By removing these constraints, a large potential improvement in cache performance can be visualized. In particular, the impact of two important cache operational concepts — reference liveness and cache bypass — to current cache design is analyzed.

It was shown that a large fraction of cache performance loss in current cache design is due to improper handling of loops with size larger than cache size and to the "cache through" constraint. By using a cache bypass bit to selectively reference directly from the main memory, a large fraction of this performance loss can be regained.

In register allocation, live range analysis has been shown to be useful in improving the effectiveness of register usage. However, this concept is completely missing in current cache management scheme. By using a liveness bit to mark those references which would become "dead" after the references are made, cache effectiveness is improved and unnecessary cache line update is eliminated.

Various hardware implementation techniques for the reference liveness cache model and/or the cache bypass model were also discussed. It is shown that modifications (both hardware and software) of current cache design to incorporate either or both of these two model are very simple; yet, cache performance improvement obtained from these two models over current cache design is tremendous. Simulations conforming this will be discussed in the next chapter.

CHAPTER VI
SIMULATION RESULTS

6.1 Introduction

In previous chapters, a complete model for managing cache using information obtained at compile time, called the MAST model, was formulated. Extensions of this model to current cache hardware design were also proposed. From the analysis, it was shown that a large improvement in system performance might possibly be obtained by using these models. Furthermore, due to hardware and software constraints in current cache design, these improvement cannot be achieved by any of the current cache management schemes.

In order to illustrate the performance improvement obtained from these compiler-driven cache management schemes and their impact on cache design, simulations of cache design using these models were performed. The main purpose of these cache simulations was not to provide some numbers about the speedup (or improvement) of a certain cache design. Rather, these simulations show that there is a large gap between performance obtained from current cache designs and that from compiler driven cache management. Moreover, this gap can only be bridged by involving program analysis in cache management schemes and releasing some *commonly-used non-beneficial* cache design constraints.

This chapter details the cache simulations performed and analyzes the results obtained from these simulations. In Section 6.2, the main objectives of these cache simulations are stated. Structures of the cache simulators used for this study are described in Section 6.3. Section 6.4 describes various parameters of these simulators. In Section 6.5, simulation results are analyzed and explanation of cache behavior using compiler-driven cache models is given whenever possible. Finally, this chapter concludes in Section 6.6.

6.2 Simulation Objectives

The main objectives in performing these cache simulations are:

- To illustrate that current cache management schemes do not perform very near the theoretical optimum (because they do not consider the static program structure).
- To show that the compiler-driven cache management schemes outlined in this thesis, which predict program behavior at compile-time by using the program's static structure, can obtain improved or even theoretically optimal cache performance.
- To evaluate the effects and importance of the various new concepts presented in this thesis, such as reference liveness and use of cache bypass.
- To re-evaluate current cache design guidelines (e.g. line size choice) using the compiler-driven cache management schemes and to support the cache design guidelines given in this thesis.
- To study the effects of various cache hardware parameters (e.g. technology of cache implementation) on the compiler-driven cache management schemes.

6.3 Simulator Structures

Although all the simulations model cache behavior across a set of rather standard benchmarks, there are actually four separate simulators. The first simulator applies the conventional LRU (least recently used) cache mechanism; the other three simulators apply the various kinds of compiler-driven cache management techniques discussed in this thesis:

LRU with Reference Liveness

This cache simulation is based on an LRU management scheme, however, a line in cache becomes "empty" as soon as all values it contains are dead. Of course, this liveness information would normally be obtained by the compiler, but the simulations use memory reference traces generated by the MIPS simulator (obtained from Stanford University), hence modifying the compiler was not convenient. Instead, the simulations computed liveness by examining the actual memory trace. This is somewhat optimistic in that it is probable that a compiler would fail to detect some of these "last references" due to

aliasing and various control-flow compilations.

LRU with Cache Bypass

This cache simulation is also based on an LRU management scheme, however, individual memory references are permitted to bypass the cache — to reference memory directly, without affecting the state of the cache. The decision of which references to bypass on should, of course, be made using analysis within the compiler. For the same reason given above, this was approximated by analyzing the actual memory trace; again, the simulation is somewhat optimistic.

The analysis used was also an approximation, however, it is a pessimistic one. Suppose that we have a particular reference, γ, to location α at a point where α is not currently in cache. Define κ as the set of references made to α after γ has placed α in cache but before α has been LRU-shifted out of cache (based on the assumption that no references bypass). Only if the total cost of loading α into cache at γ plus making the references of κ from the cache is greater than the cost of making the references of κ directly from memory, then reference γ will bypass the cache. Obviously, if α is currently in cache (given the bypass decisions made thus far), then γ will reference the cache.

MAST Model

This cache simulation is not based on an LRU management scheme, rather, it assumes that the compiler is able to determine not only whether each reference should go through or bypass the cache, but also that the compiler may specify where within a cache set the line should be placed if it is to be placed in cache. The full MAST model, as proposed in Chapter 3 and 4 of this thesis, is applied. Since, as above, we have performed the analysis on the memory reference trace (rather than on the source program), the MAST model is given perfect information; the MAST model literally finds the optimal cache management given a set of constraints, hence, this simulator gives the exact, true, absolute upper bound on cache performance. Again, this is somewhat better than would be expected if the compiler were used to analyze the source code.

It is somewhat unfortunate that the usual basis for comparing cache techniques is the evaluation of performance for memory traces, because this makes the proposed techniques seem somewhat more effective than they actually are. However, it is very difficult to compare *only* cache technique if

one were to create a new/modified compiler to implement the proposed compiler-driven management techniques; for example, live range analysis would significantly alter the way the compiler allocates memory, hence, more than just the cache management would be likely to change. Further, experience with sophisticated compilers suggests that compiler technology can be expected to provide nearly perfect information most of the time — exact information can probably be obtained for at least 90% of all references[1].

6.4 Simulation Parameters

Each set of simulations was run varying a number of cache hardware parameters, so that the effects of such changes on the various management policies could be determined. These parameters were:

memory reference trace

There were four different programs, compiled and executed by the Stanford University MIPS simulation package, which were used. Because memory reference traces easily become unmanageably long, only the trace from executing the first half million instructions were considered. (This is long enough to make "cold start" cache effects negligible.) The source programs were also taken from the MIPS package, and are widely used as benchmarks of cache and/or system performance:

Intmm

A program which performs a matrix multiplication of two integer matrices, each of which is 40 by 40.

Bubble

A typical bubble sort program, executed on a set of 500 random data.

Puzzle

This is a compute-bound program from Forest Basket, run with a size of 511.

---

[1] This is an approximate number based on the assumption that *all* pointer references confuse the compiler's analysis and that the statistics given in [Mil88] are at least approximately correct.

Tower

The standard recursive tower-of-Hanoi solution, given the problem of moving 18 disks.

mode

This describes the memory reference modes during which the cache may be active. If the cache may be active only during an instruction fetch, the cache is in "inst." mode. If the cache may be active only during a data store/load, the cache is in "data" mode. A cache which may be used for both instruction and data references is in "mix" mode.

cache size

Total number of words which may be held in the cache. The range of cache sizes simulated was 32 to 1024 words.

lines/set

The maximum number of cache lines which may be associatively examined when a reference to a particular memory address is made through the cache. Various power-of-2 values were simulated.

words/line

The number of words which may be held in a single cache line; also, the number of words transferred between main memory and the cache whenever such a transfer is performed. Various power-of-2 values were simulated.

relative delay

This is the ratio of main memory access time to cache access time. Typical values for this differ widely depending on the hardware implementation and fabrication technology. Values of 2, 5, and 10 are used in the simulations.

burst cost

Given a cache with a line size (words/line) of λ, the burst cost is a percentage indicating the time ratio between accessing $\lambda - 1$ words as individual bypass references versus the cost of moving one line between memory and cache minus the cost of one word being accessed via cache bypass. In other words, it is the fraction of the bypass cost which is incurred for each word after the first in a burst which is the transfer of a cache line; it is assumed that the transfer of the first word takes the same time in either case. Values of 0%, 25%, 50%, 75%, 90%, and

100% are used in the simulations.

Suppose S_{line} is the line size of the cache, F_{saving} is the saving factor (defined as 100% - burst cost) and T_{access} is the relative delay. Various costs of information transfer and reference are defined as follows:

The cost of referencing a word from cache is:

$$Cost_{cache} = 1$$

The cost of referencing a word directly from main memory (i.e. using cache bypass option) is:

$$Cost_{main\_memory} = T_{access}$$

The cost of placing a line in cache is:

$$Cost_{placement} = T_{access} * (1 + (S_{line} - 1) * (1 - F_{saving}))$$

The cost of replacement of a "dirty" line in cache is:

$$Cost_{placement} = 2 * T_{access} * (1 + (S_{line} - 1) * (1 - F_{saving}))$$

6.5 Simulation Results and Discussion

A total of more than 30,000 cache simulations were performed, encompassing a wide variety of cache configurations, cache management schemes, and system architectures. Clearly, results obtained from these simulations are too large to all be presented in this thesis. Since these simulation results are very consistent, we have chosen some representative ones to graph and present here.

To make the simulations as complete as possible, all possible power-of-2 cache organizations (e.g. different line sizes, set sizes) for each cache size were simulated. Cache simulations for each of the three modes of cache usages — data only, instruction only, and both data and instruction — were also performed for each of these cache organizations. The range of cache size simulated was from 32 to 1024 words; the values of relative delay used for simulations were 2, 5, and 10; and the values of saving factor used for simulations were 0%, 10%, 25%, 50%, 75% and 100%.

Figure 6.1 to Figure 6.9 graph the speedup of total reference times of the integer matrix multiplication program with cache bypass model, reference liveness model or the MAST model as compared to the same configuration conventional cache using LRU model. A cache with size 64

words, a relative delay of 10, and a saving factor of 25 percent was used and each of the three modes of cache usages was simulated. Each curve in the graphs is marked with the power-of-2 which was used as the associative set size. For example, an "*3*" on a curve means an set associativity of eight.

These curves clearly show that the speedup of total reference times using the cache bypass model and the MAST model are very large — in fact, it is plotted on a *log scale*, average about 2, and range from 1.1 to 40. For the reference liveness model, even though the speedup is much smaller than the bypass and MAST model, it still ranges from a few percent to 20 percent.

This phenomenon of cache performance improvement by using these three cache management schemes is expected. Since the concept of live range of references is missing in current cache management schemes, some performance is lost. However, the principles of locality of references do provide a good guidelines of what should be kept in cache. Consequently, the speedup of total reference times using the reference liveness model is from a few percent to about 20 percent.

On the other hand, the selective cache bypass concept is completely missing in current cache design. This is somewhat even opposite to current cache design philosophy that it is always good to place information in cache. As a result, a significant fraction of cache performance is lost due to the "cache-through" constraint.

Although the MAST model provides the best cache performance among the three new models, the difference between performance from the MAST model and that from the cache bypass model is not very large. This is because the most important concept missing in current cache design — placing information in cache only if it can help reducing total reference times — is already in these two models.

With an increase in cache line size (leaving cache size and cache set size fixed), the relative speedup of total reference times using the reference liveness model decreases. Since a cache line is defined to be dead in the simulator if and only if all elements in a cache line are dead, an increase in cache line size would reduce the probability for a cache line to be dead because more information needs to be dead simultaneously. Hence, the effect of the reference liveness model becomes smaller.

In the cache bypass model and the MAST model, an increase in cache line size (leaving cache size and cache set size fixed) would greatly increase

the relative speedup of total reference times. This agrees and confirms the argument given in previous chapters. A larger line size implies a larger overhead in cache line placement and replacement. Although the total number of references of a line with increasing line size increases, this increase is much less than the increase in overhead. Consequently, cache more easily becomes polluted, and the cache bypass concept becomes more critical in improving system performance.

By comparing Figure 6.1 to Figure 6.3, Figure 6.4 to Figure 6.6, and Figure 6.7 to Figure 6.9, the effects of different modes of references to the cache performance improvement in the three new models can be seen. Cache performance improvement is the largest for data only cache. This is probably due to the fact that data is more randomly accessed and current cache management schemes, which are mostly based on the sequential reference feature, are not good at managing data cache efficiently. For instruction cache, cache performance improvement is the smallest. This is due to the better management of instruction cache using the principle of locality of references.

Figure 6.10 to Figure 6.18 graph the total reference times of the integer matrix multiplication program with the cache bypass model, reference liveness model or the MAST model as compared to the same configuration conventional cache using LRU model. The parameters used in these nine graphs were the same as those in Figure 6.1 to Figure 6.9. A cache with size 64 words, a relative delay of 10, and a saving factor of 25 percent was used and each of the three modes of cache usages was simulated. Each curve in the graphs is marked with the power-of-2 which was used as the associative set size. The dotted lines indicate the times taken using conventional cache and LRU whereas the solid lines show the times taken with one of the three new cache models.

Aside from the obvious cache performance improvement in using the three new cache models, these graphs suggest some interesting general cache design rules. First, if the total memory reference time is to be minimized, rather than the cache hit-ratio maximized, it is usually better to choose small line size and small set size. This makes perfect sense in that although large line sizes increase hit-ratio, they imply overhead increases which are greater than the hit-ratio increases — in fact, exponentially greater. As a result, the total reference times increases as the cache line size increases.

Second, for the cache bypass model or the MAST model, the difference in total reference times for different line sizes (with same cache size and set size) is not as great as those for cache with conventional LRU model or the reference liveness model. This is true because a lot of cache pollution in LRU or the reference liveness model can be avoided by considering the relative benefit of placing a line in cache and referencing those "cache polluting information" directly from main memory.

Figure 6.19 to Figure 6.21 graph the relative cache performance using LRU, reference liveness, cache bypass and the MAST model for different line sizes. A cache with size 64 words and set size of 4, a relative delay of 10, a saving factor of 25 percent was used and the benchmark program simulated was the integer matrix multiplication. Each of the three modes of cache usages was simulated. The curve marked with "*l*" is the one using LRU; the curve marked with "*v*" uses the reference liveness model; the curve marked with "*b*" uses the cache bypass model; and the curve marked with "*m*" is the one using the MAST model.

The effectiveness of these four cache models to cache performance is very clear: the MAST model gives the best performance, the cache bypass is the next, followed by the reference liveness, and the one with relatively worst performance is the LRU. Further, the difference in cache performance between the MAST model and the cache bypass model is relatively small when it is compared with that between the cache bypass model and the reference liveness model. And the performance difference between the LRU and the reference liveness model is again relatively small. All these agree with the observation from Figure 6.1 to Figure 6.9 and with the explanation given in the beginning of this section.

Figure 6.22 to Figure 6.24 repeated the same experiment of Figure 6.17 except that other benchmark programs such as bubble, puzzle or towers was used instead of the integer matrix multiplication program. These graphs show that the expected improvement of system performance using any of the three new cache models is quite independent of reference programs. This is true because no assumption about program reference characteristic is made in all these cache models. The rapid increase in total reference times with increasing cache line size is also shown in these three graphs.

Figure 6.25 to Figure 6.28 plot the total reference times vs. the cache hit ratio for a fixed cache with size 64 words, a relative delay of 10, a saving factor of 25 percent and a "mixed" cache usage mode. All possible cache

configurations (different cache line size and set size) with the given cache sizes were simulated for each of the four cache models. Each point "X" in these figures represents one possible cache configuration. Note that a cache bypass reference (i.e. reference made directly from main memory) does not consider a cache hit nor a cache miss.

These graphs show one significant result: cache hit ratio is not necessarily related to the total reference time. With higher cache hit ratio, the total reference times might increases instead of decreases. Although this observation might seem to be a big surprise to cache designers, it actually makes more sense. Cache hit ratio only indicates the probability of finding information in cache. However, it does not consider the penalty of each cache miss. Since a change in cache configuration (e.g. increase cache line size) would probably change the penalty of a cache miss, comparison of cache performance of different cache configurations using cache hit ratio makes sense only if the cache miss penalty in each of the compared configurations are the same. Otherwise, a configuration with lower hit ratio and lower overhead of line placement might perform better than one with higher hit ratio and higher line placement overhead.

Figure 6.25 and Figure 6.26 show that the relationship between the total reference times and the cache hit ratio in the LRU or the reference model is somewhat random. Moreover, the cache hit ratio can range from a low value (e.g. 0.4) to a high value (e.g. 0.95). Although the relationship between the total reference times and the cache hit ratio using the cache bypass or the MAST model is still random, the range of possible cache hit ratios is much smaller (e.g from 0.7 to 0.95).

This phenomenon can be explained by the fact that a lower cache hit ratio in the LRU or the reference liveness model always implies more cache lines with low reference frequencies. In the cache bypass model or the MAST model, references to these lines with low reference frequencies would bypass the cache. Hence, only those lines with high reference frequencies are placed in cache and only their references are counted towards the total number of cache hits and misses. As a result, the cache hit ratio would not be too low.

Figure 6.29 graphs the total reference times of the integer matrix multiplication program for different cache saving factors. A cache with size 64 words, set size of 4, line size of 8. and a relative delay of 10 was used and the cache is in the "mixed" usage mode. Each of the four cache models

were also simulated. Again, an "*l*" implies the LRU model; an "*v*" implies the reference liveness model; an "*b*" implies the cache bypass model; and "*m*" implies the MAST model.

This graph shows that increasing saving factor decreases the total reference times linearly in the LRU or the MAST model. This is true because the cache miss penalty is a linear function of the saving factor. Although increasing saving factor also decreases the total reference time in the cache bypass model and the MAST model, the curves are *not* straightly linear. This is because as the saving factor changes, the overhead of placing a line in cache changes and this will affect the decision of bypassing the cache. These four curves converge because increasing saving factor causes more lines to be placed in cache before they are referenced; hence, the effect of the cache bypass option in the cache bypass model and the MAST model becomes less. Note that the curves for the cache bypass model and the MAST model curve slightly towards the X-axis.

Figure 6.30 to 6.32 graph the total reference times of the integer matrix multiplication for different saving factors and different cache line sizes. A cache with size 64 words, set size of 4, and a relative delay of 10 was used and it was in the "mixed" usage mode. In each of these three graphs, LRU was used for comparison with each of the three new cache models. The dotted lines indicate the times taken using conventional cache and LRU whereas the solid lines show the times taken by one of the three new cache models. Each curve in the graph is marked with a number which was used as the cache line size.

Aside from the obvious relationship between the total reference times and the saving factor, these graphs also indicate the effect of cache line size to this relationship. With a larger cache line size, the rate of decreasing total reference times with increasing saving factor is much higher than that with smaller cache line size. This can be explained by the fact that the cache miss penalty is a linear function of the product of saving factor and the cache line size. Consequently, the slope of the curves in these three graphs is a linear function of the cache line size and increasing cache line size increases the rate of decreasing total reference times with increasing saving factor. Note than when the cache line size is one, the total reference times is not affected by the saving factor. This is because the time to transfer the first word of a line is independent of the saving factor.

Figure 6.33 repeated the experiment of Figure 6.29, except that the total reference times was graphed vs. different relative delays instead of different saving factors. Again, a cache with size 64 words, set size of 4, line size of 8, and a saving factor of 25 percent was used and the cache was in the "mixed" usage mode. Each of the four cache models were simulated and the symbols in these graphs were the same as those in Figure 6.29.

This graph shows that increasing the relative delay increases the total reference times linearly in the LRU and the reference liveness model. This makes sense because the cache miss penalty is a linear function of the relative delay. In the cache bypass model or the MAST model, the relationship between the total reference times and the relative delay is not straightly linear because changing the relative delay might change the decision of bypassing the cache of a reference. With an increase in relative delay, the overhead of placing a line in cache increases, causing more references to bypass the cache in the cache bypass model or the MAST model. Consequently, the curves of the cache bypass model and the MAST model curve slightly towards the X-axis.

Figure 6.34 to Figure 6.36 repeat the experiments of Figure 6.30 to Figure 6.32, except that the total reference times was graphed vs. different relative delays instead of different saving factors. A cache with size 64 words, set size of 4 and saving factor of 25 percent was used and the cache was in the "mixed" usage mode. In each of the three graphs, LRU was used for comparison with each of the three new cache models and the meaning of the symbols used were exactly the same as those in Figure 6.30 to Figure 6.32.

These graphs show that the rate of increasing total reference times with increasing relative delay is higher for cache with larger cache line size. This is true because cache miss penalty and the cost of referencing directly from main memory is directly proportional to cache line size, implying that the slope of curves in these three graphs is a function of the cache line size.

6.6 Conclusions

In this chapter, simulation results of those new cache models discussed in the last few chapters were presented and analyzed. It was shown that these new cache models — reference liveness model, cache bypass model, and the MAST model — do give the expected improvement in cache performance. This shows how cache performance can potentially be greatly improved by incorporating program reference behavior and control flow analysis into cache management schemes with the help of the compiler.

The effects of various cache parameters such as cache line size, saving factor, and the relative delay to the total reference times were also studied. Explanations for these phenomena were also given and guidelines for cache designs using compiler-driven cache management were proposed.

Aside from the performance improvement from these new cache models, some of the important observations from these cache simulation results can be summarized as follows:

- The cache hit ratio is not necessarily related to total reference times.
- Small cache line size and set size are usually preferred if the total reference times is the parameter being optimized.
- One of the most important concept missing in current cache management schemes is the selective cache bypass and a large fraction of performance loss in current cache management schemes is due to the "cache-through" constraint.

After applications of the MAST model and its extensions to cache management schemes are made, the next chapter will discuss how the same model can be applied to register allocation.

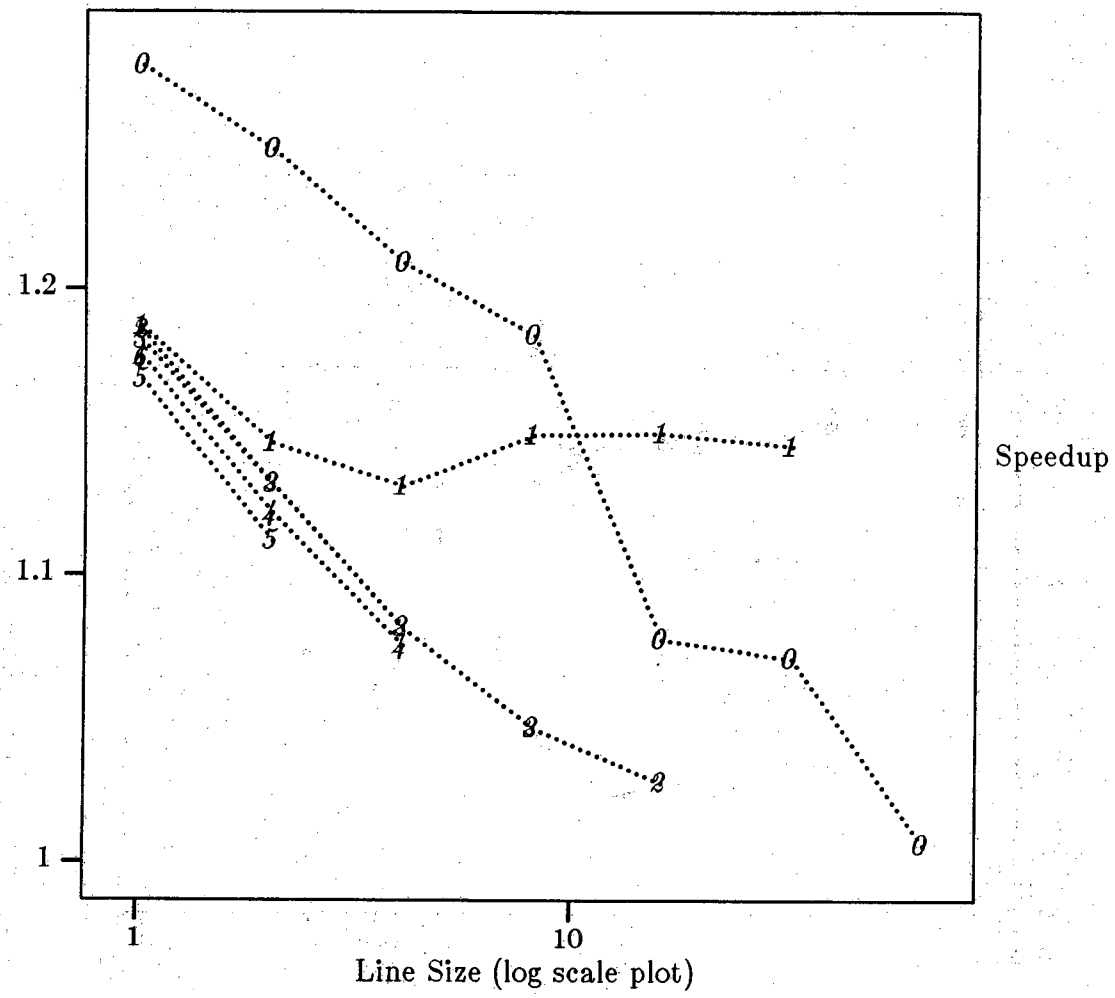


Figure 6.1: Speedup of Liveness Model Over LRU Using Intmm, Data Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

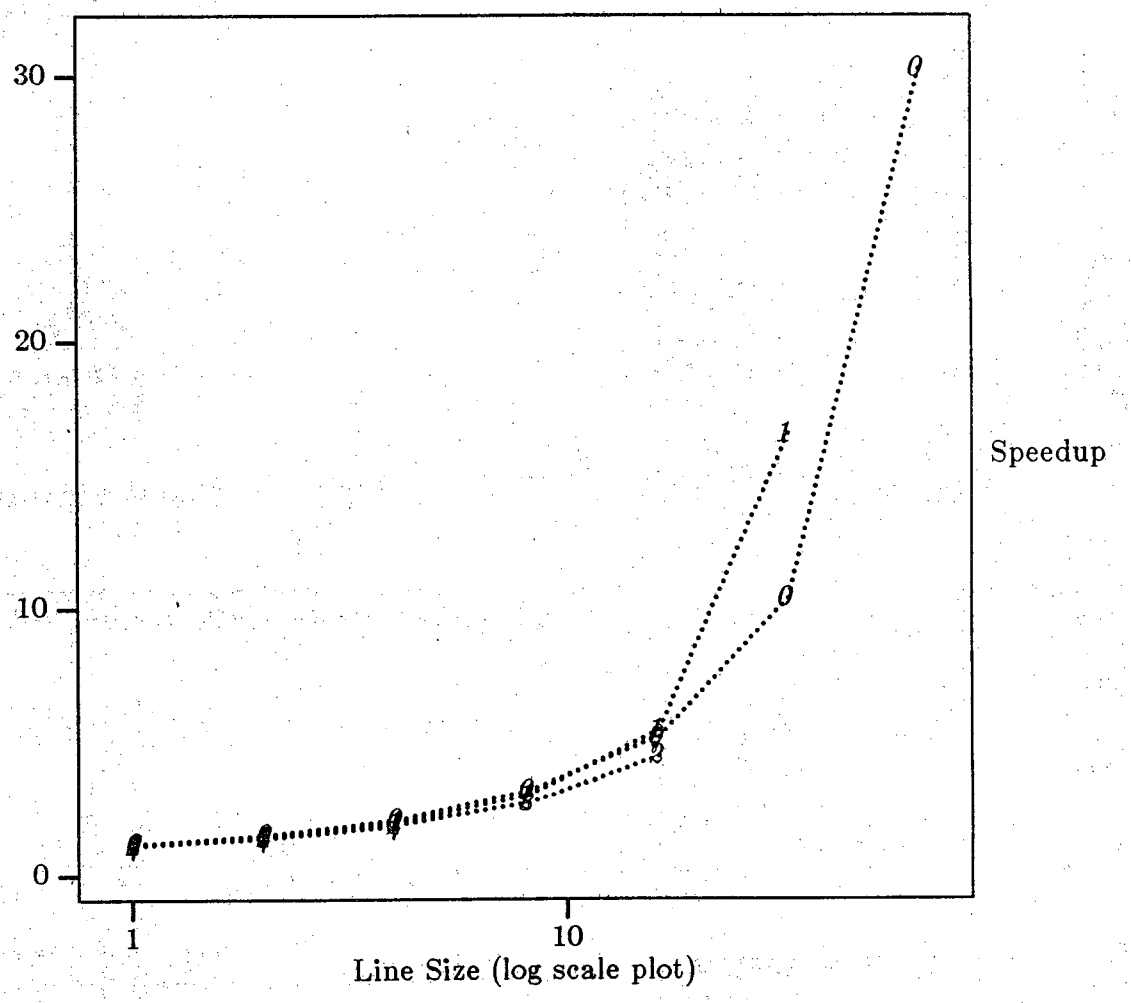


Figure 6.2: Speedup of Bypass Model Over LRU Using Intmm, Data Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

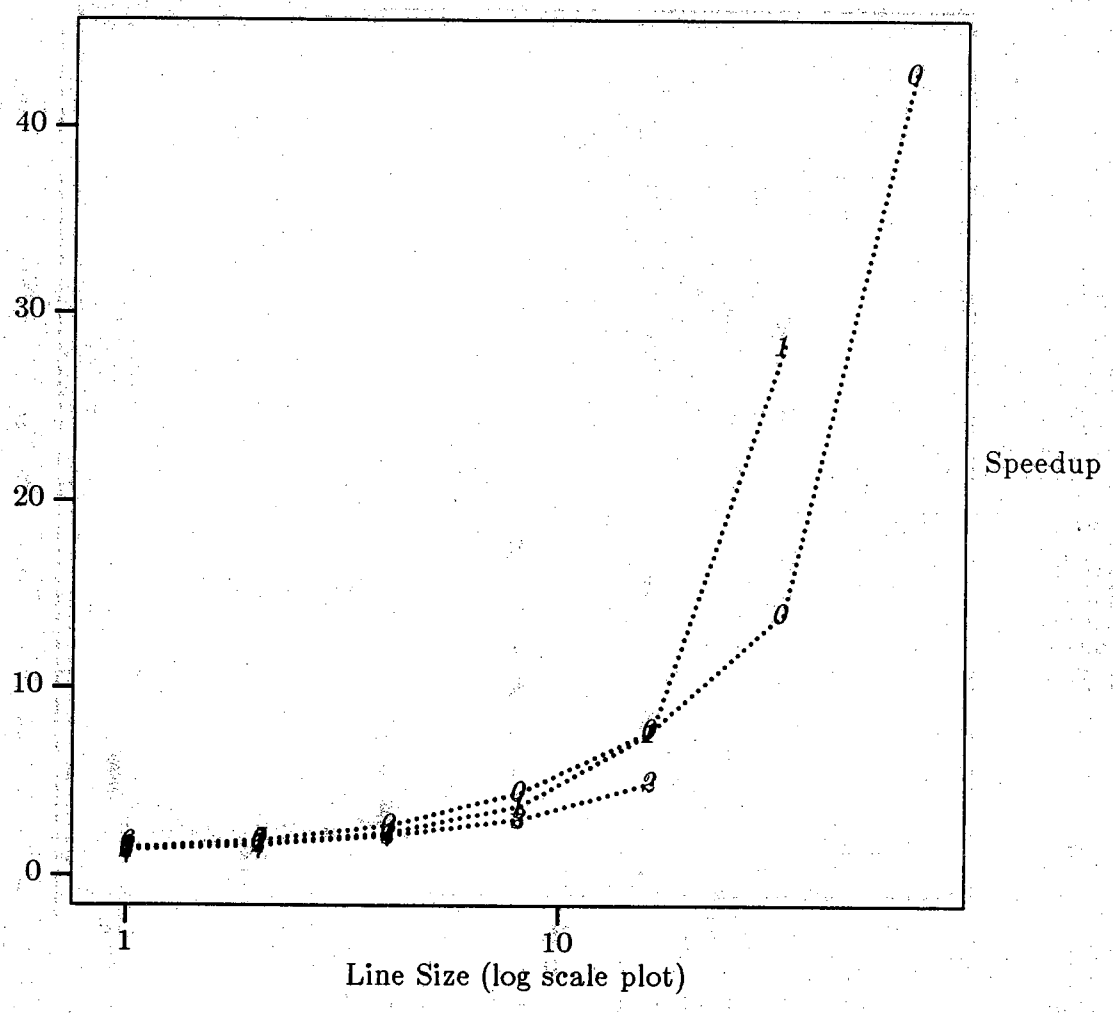


Figure 6.3: Speedup of MAST Model Over LRU Using Intmm, Data Cache of Size 64, Access Ratio of 10 and Saving Factor of 25%

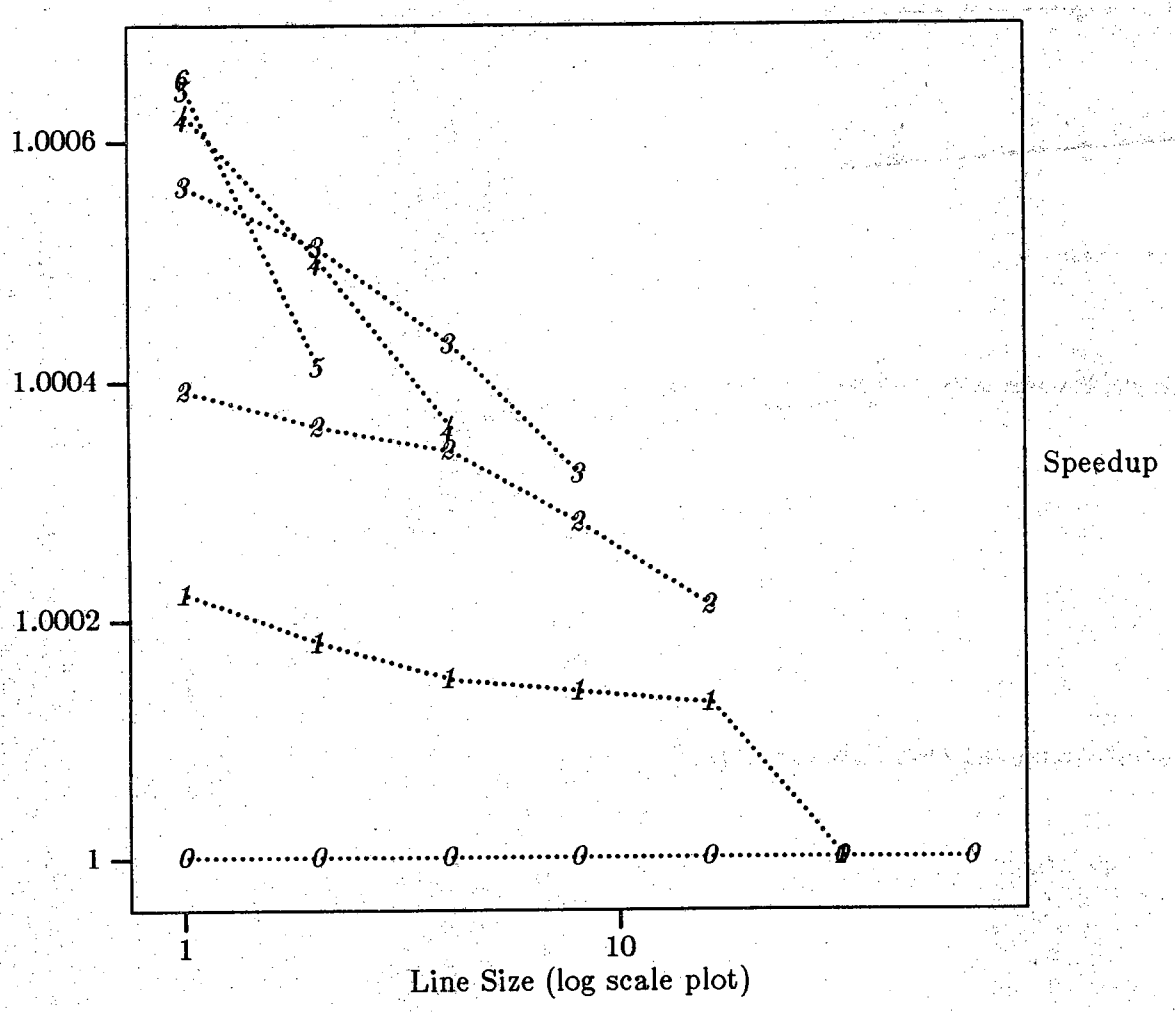


Figure 6.4: Speedup of Liveness Model Over LRU Using Intmm, Instruction Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

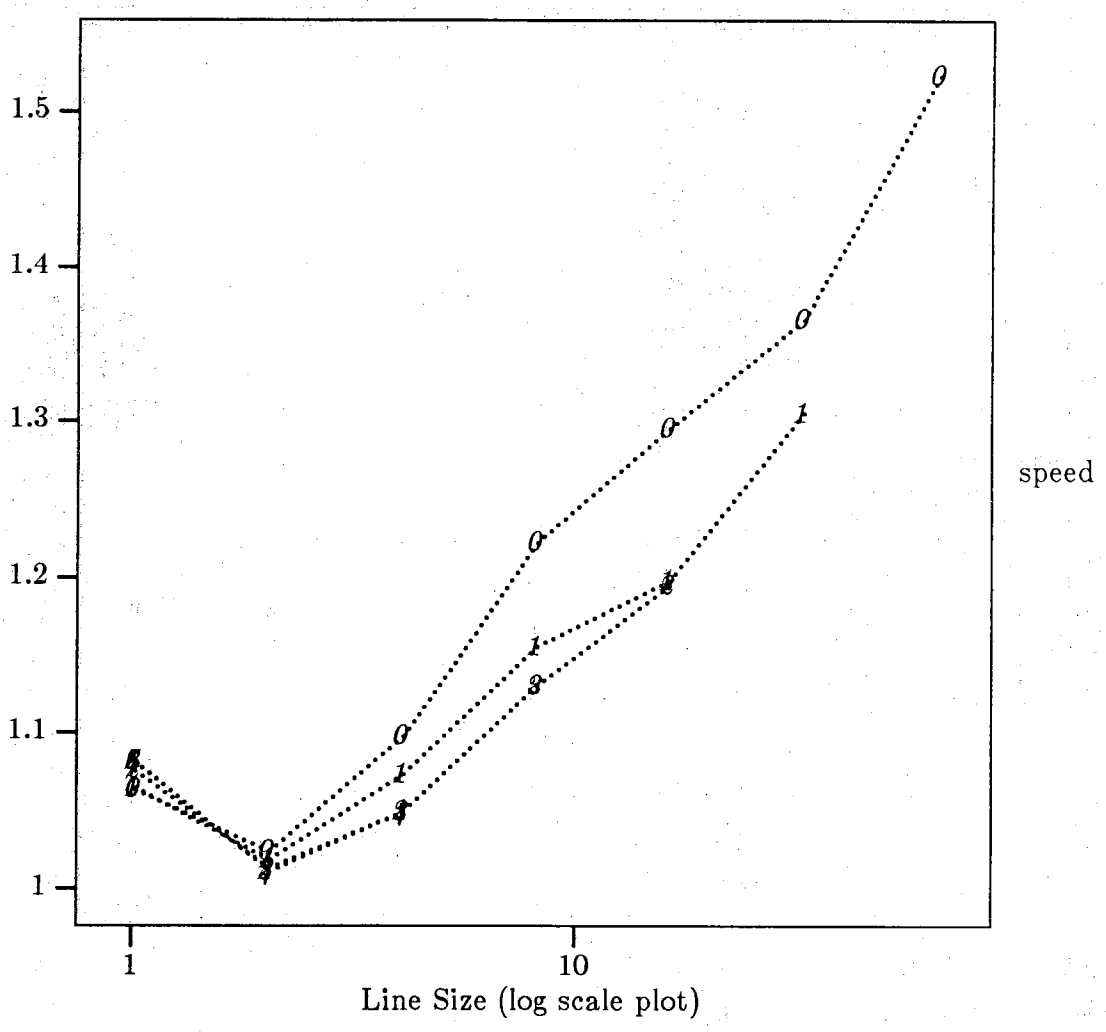


Figure 6.5: Speedup of Bypass Model Over LRU Using Intmm, Instruction Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

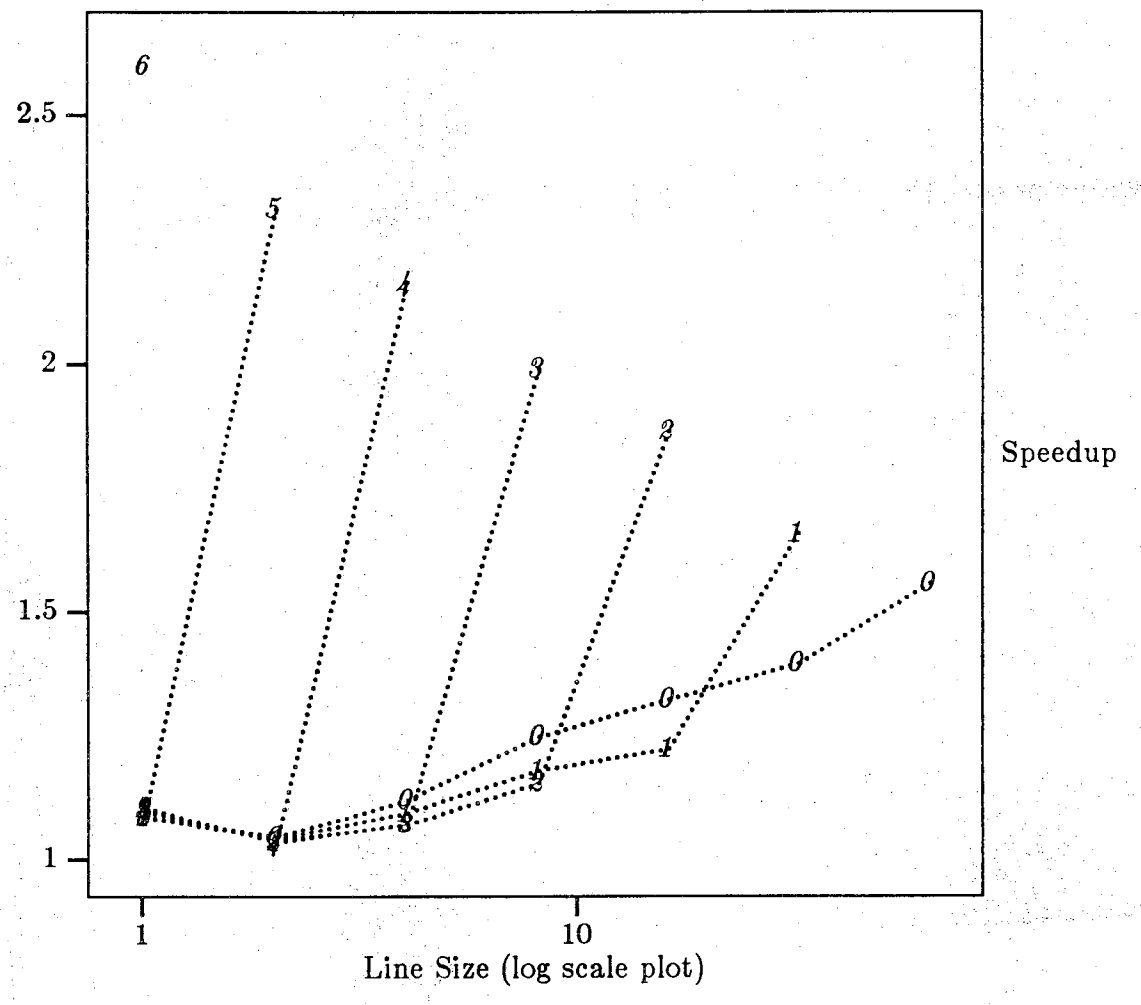


Figure 6.6: Speedup of MAST Model Over LRU Using Intmm, Instruction Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

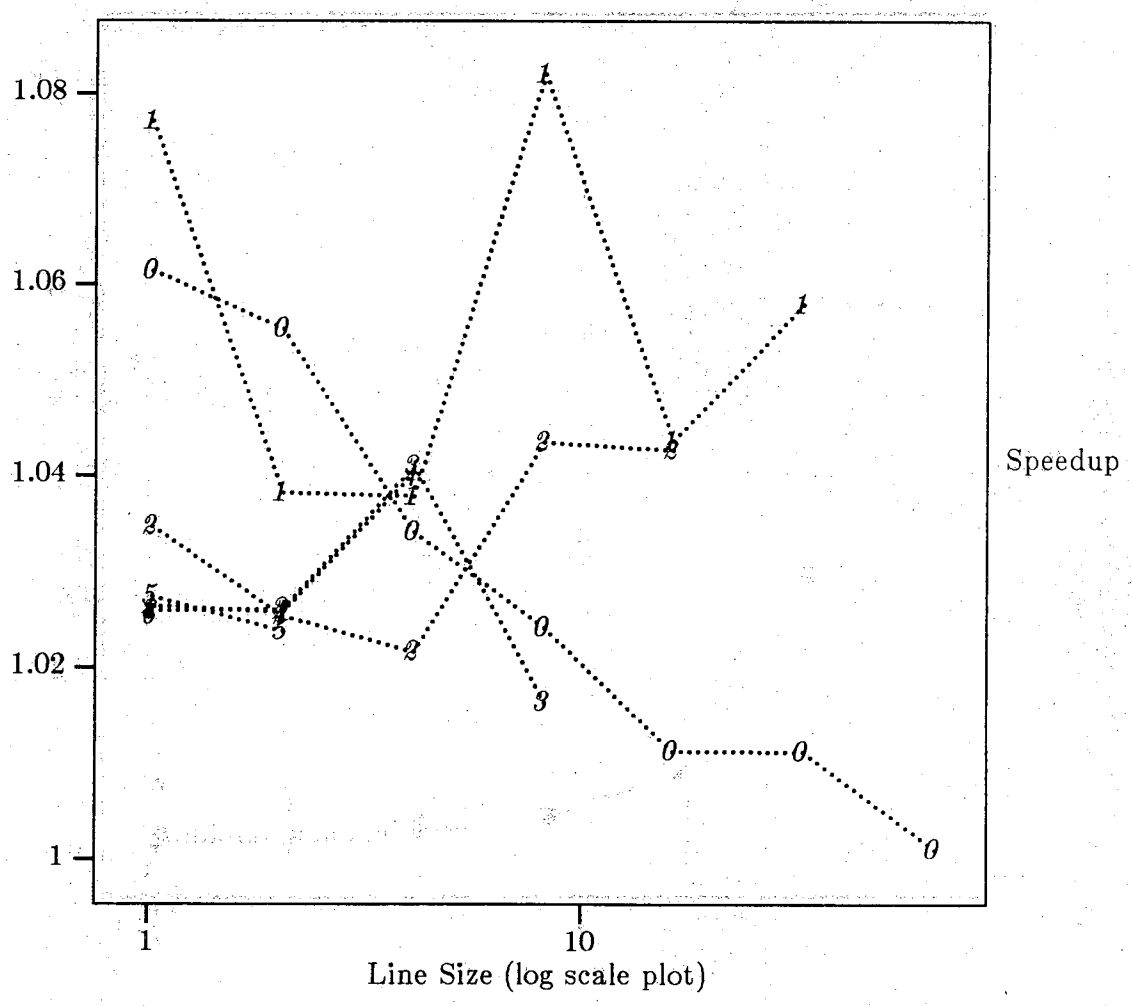


Figure 6.7: Speedup of Liveness Model Over LRU Using Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

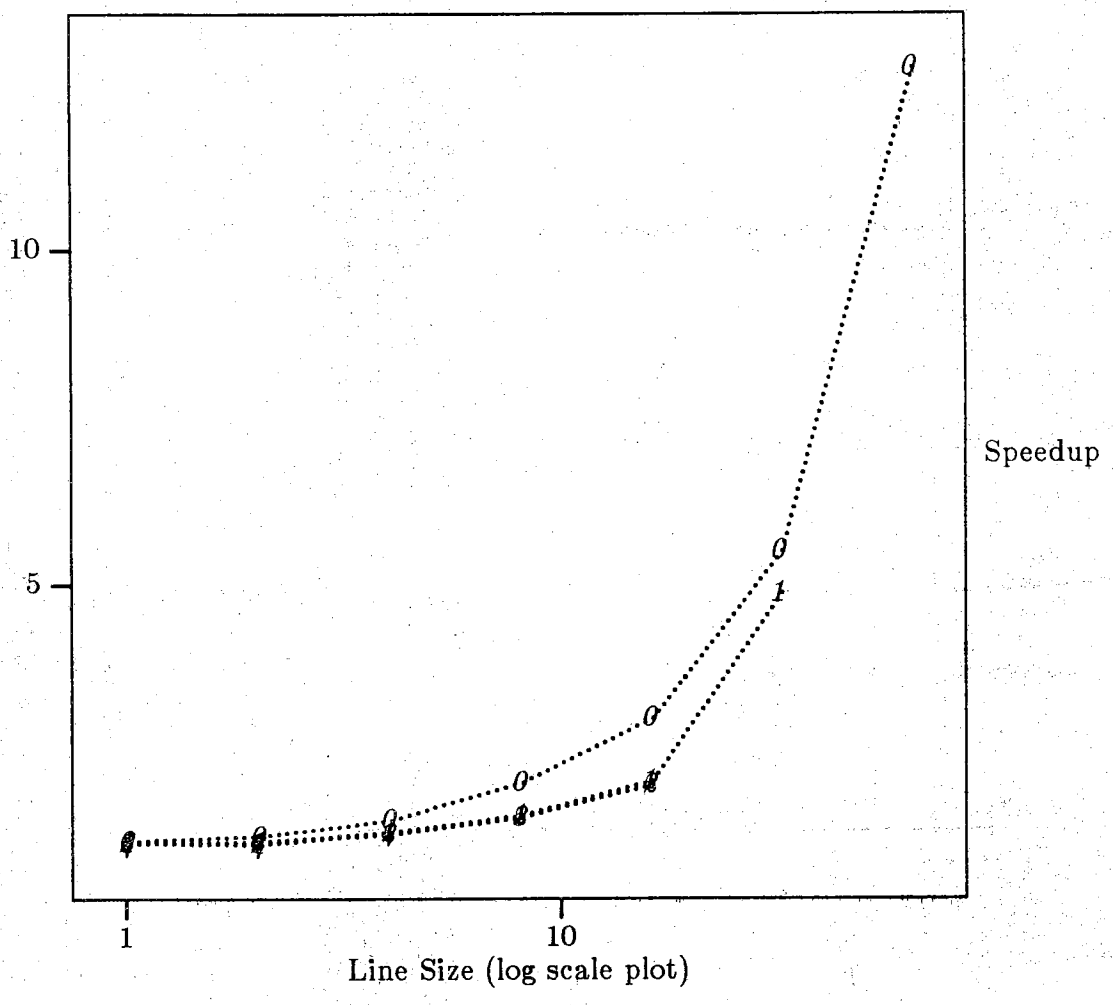


Figure 6.8: Speedup of Bypass Model Over LRU Using Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

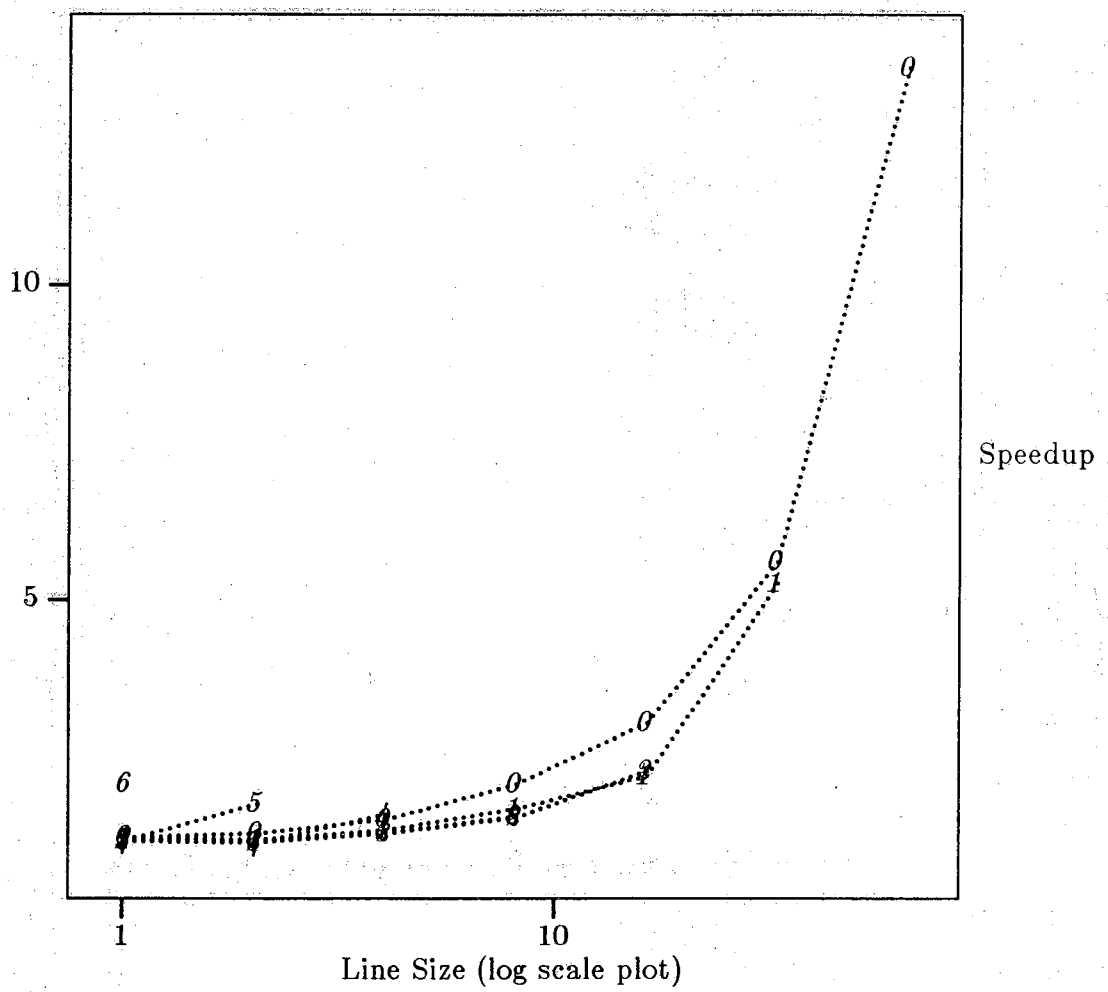


Figure 6.9: Speedup of MAST Model Over LRU Using Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

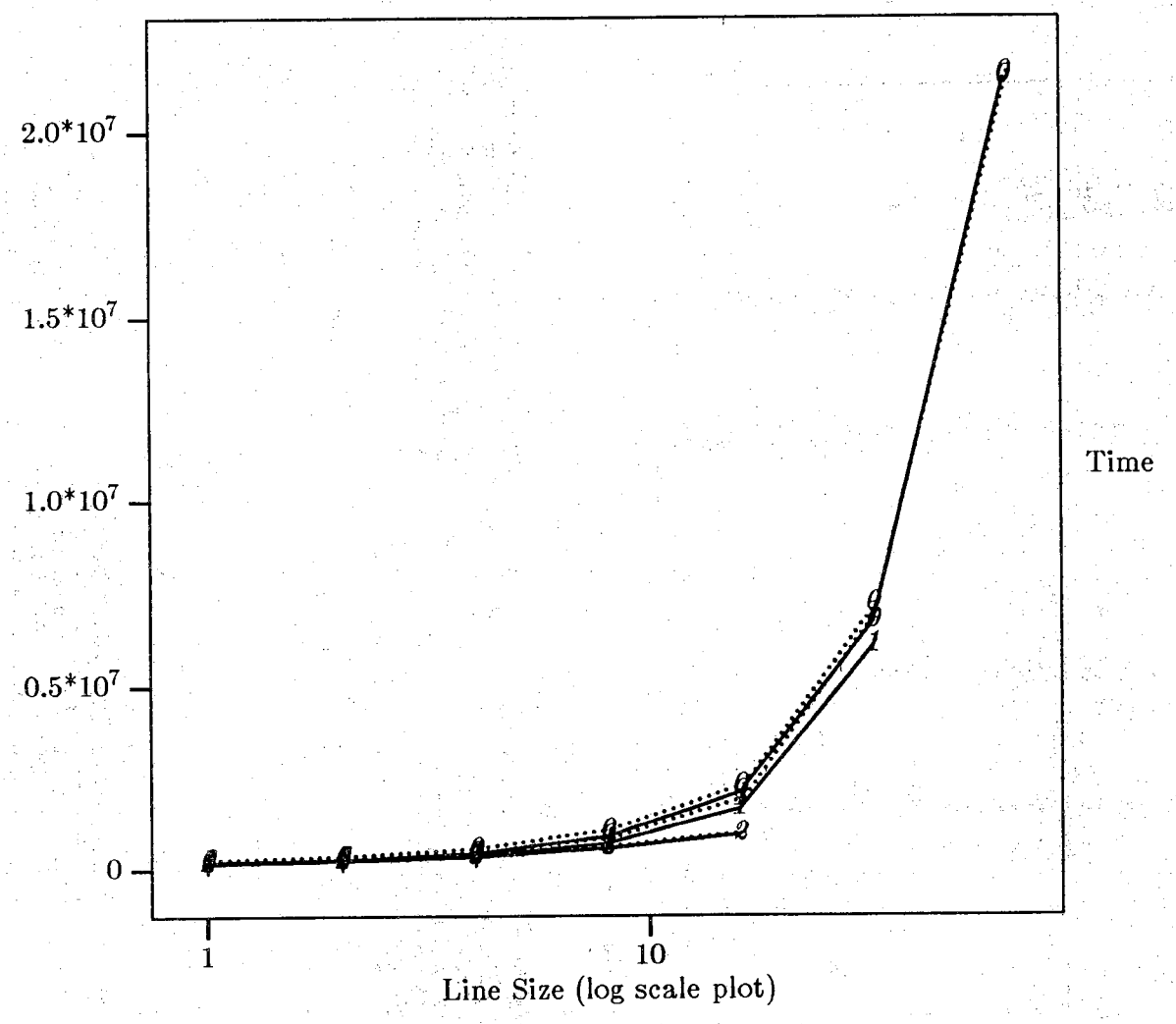


Figure 6.10: Reference Time of Liveness Model and LRU Using Intmm, Data Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

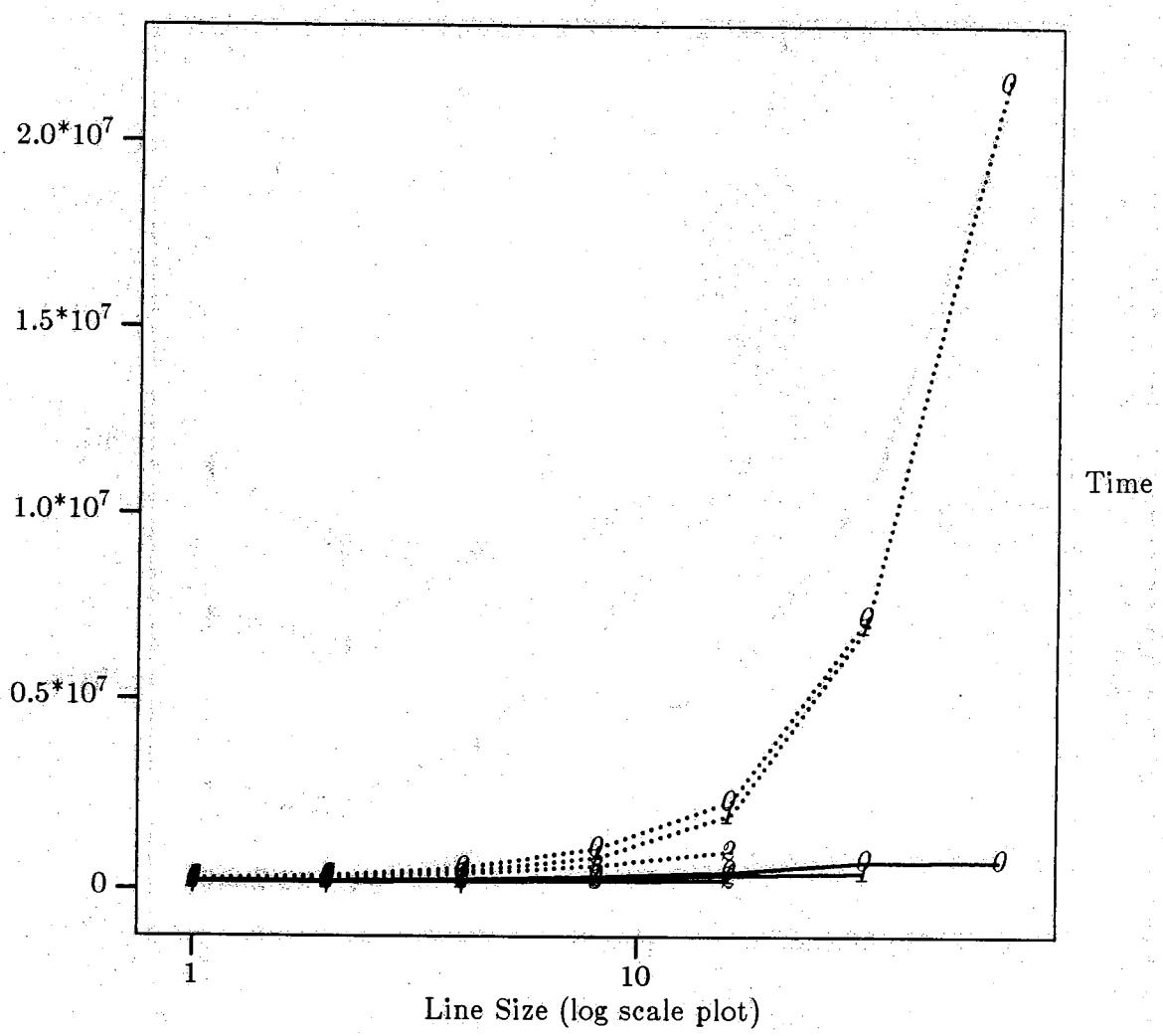


Figure 6.11: Reference Time of Bypass Model and LRU Using Intmm, Data Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

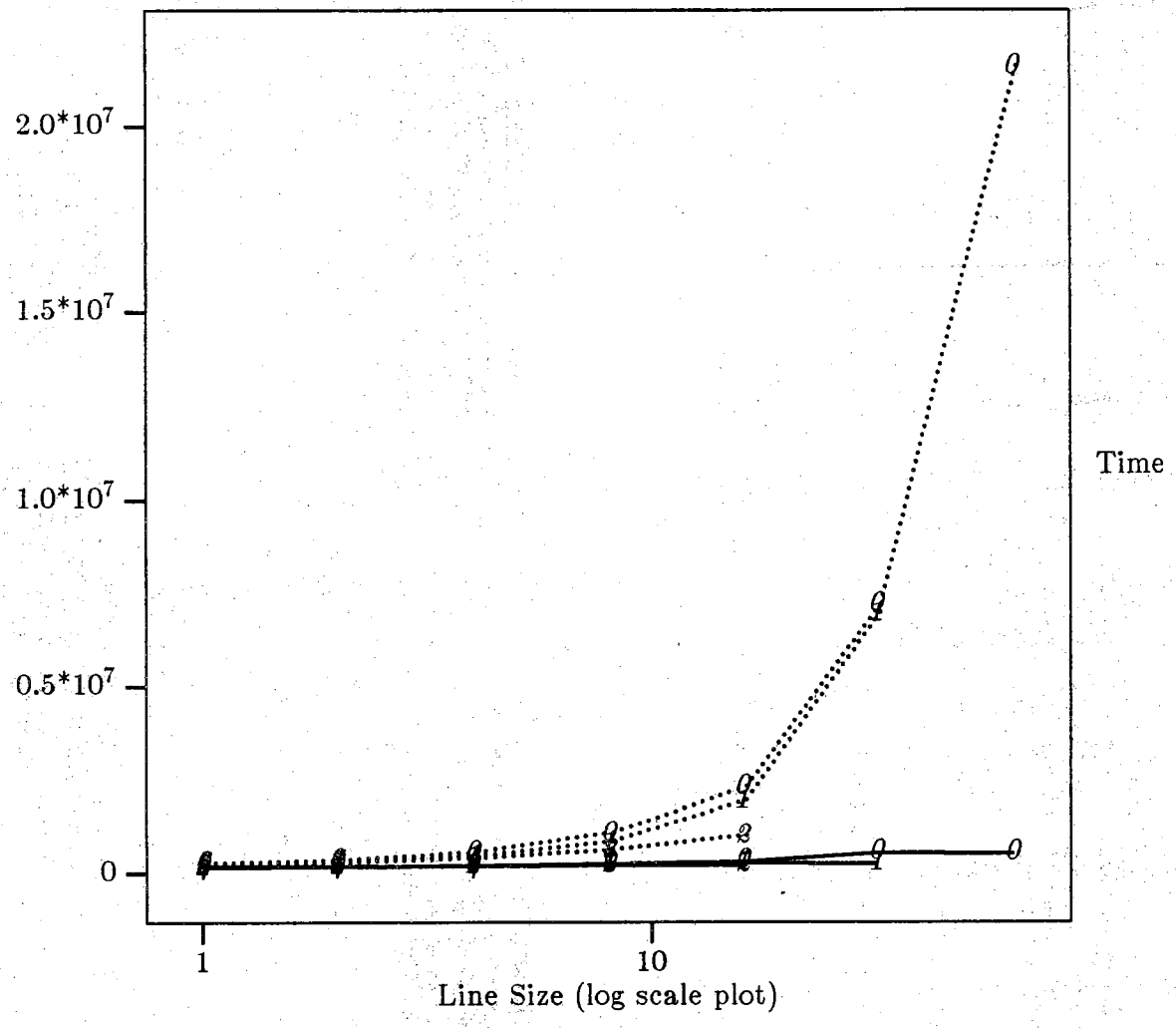


Figure 6.12: Reference Time of MAST Model and LRU Using Intmm, Data Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

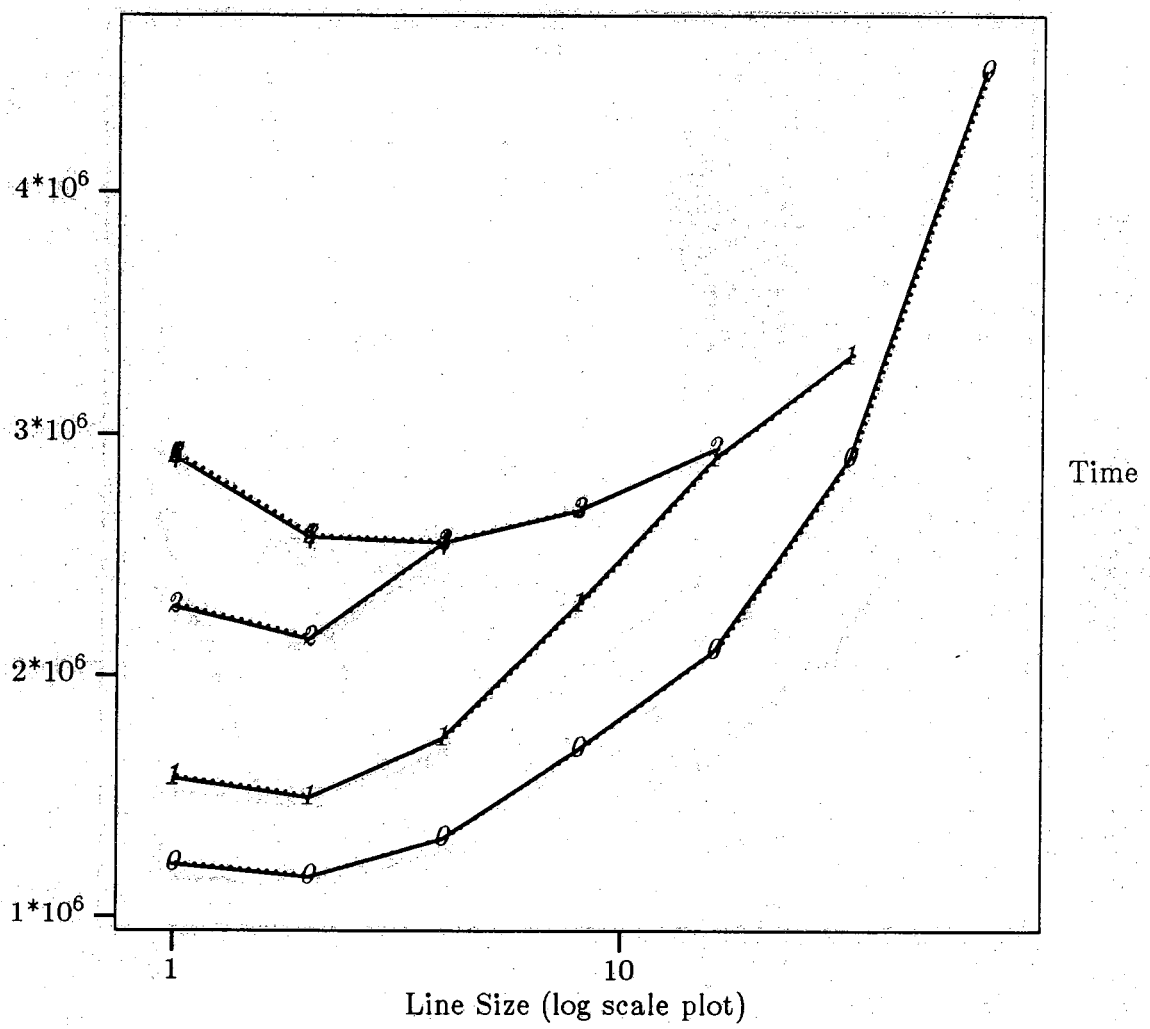


Figure 6.13: Reference Time of Liveness Model and LRU Using Intmm, Instruction Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

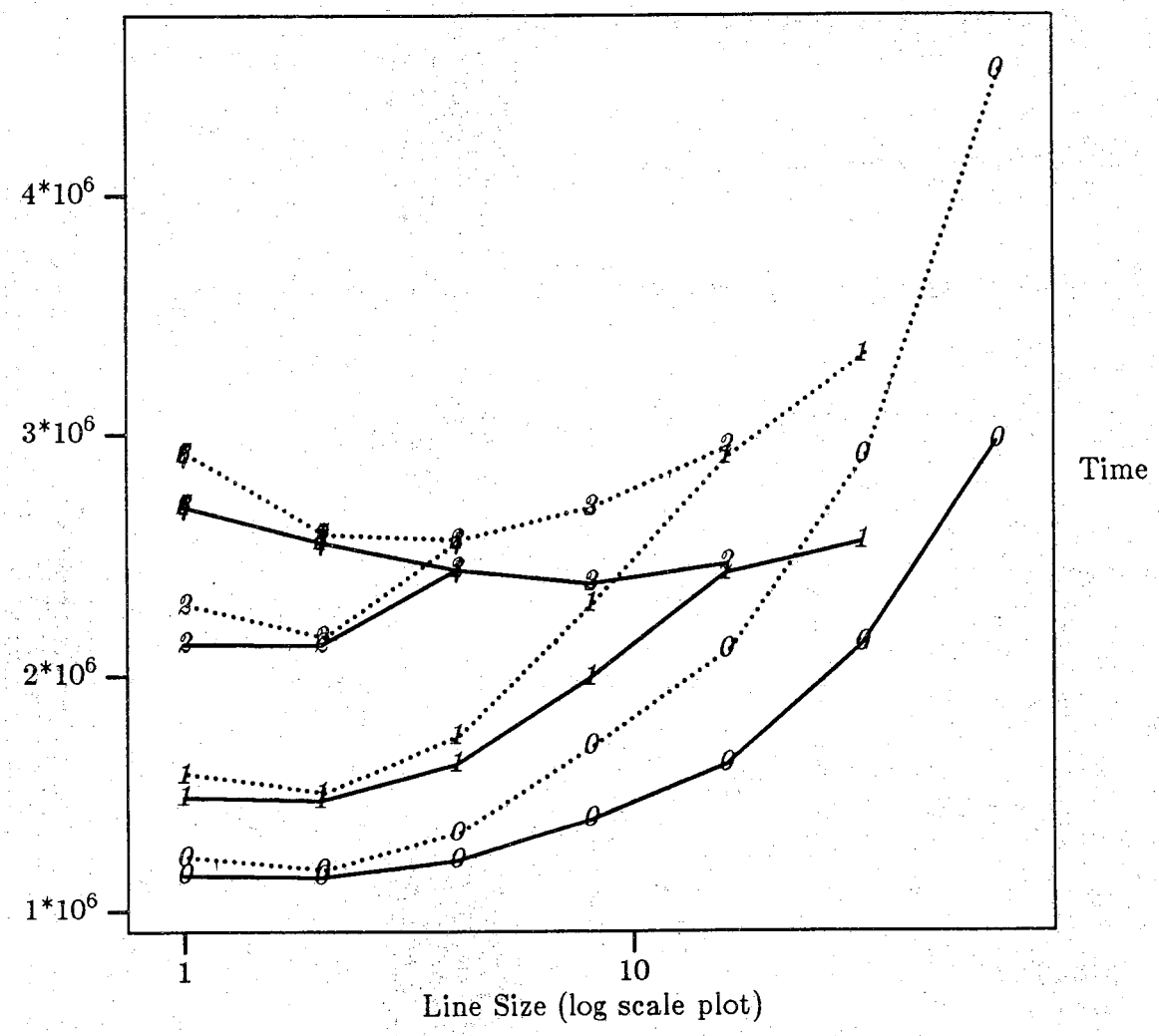


Figure 6.14: Reference Time of Bypass Model and LRU Using Intmm, Instruction Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

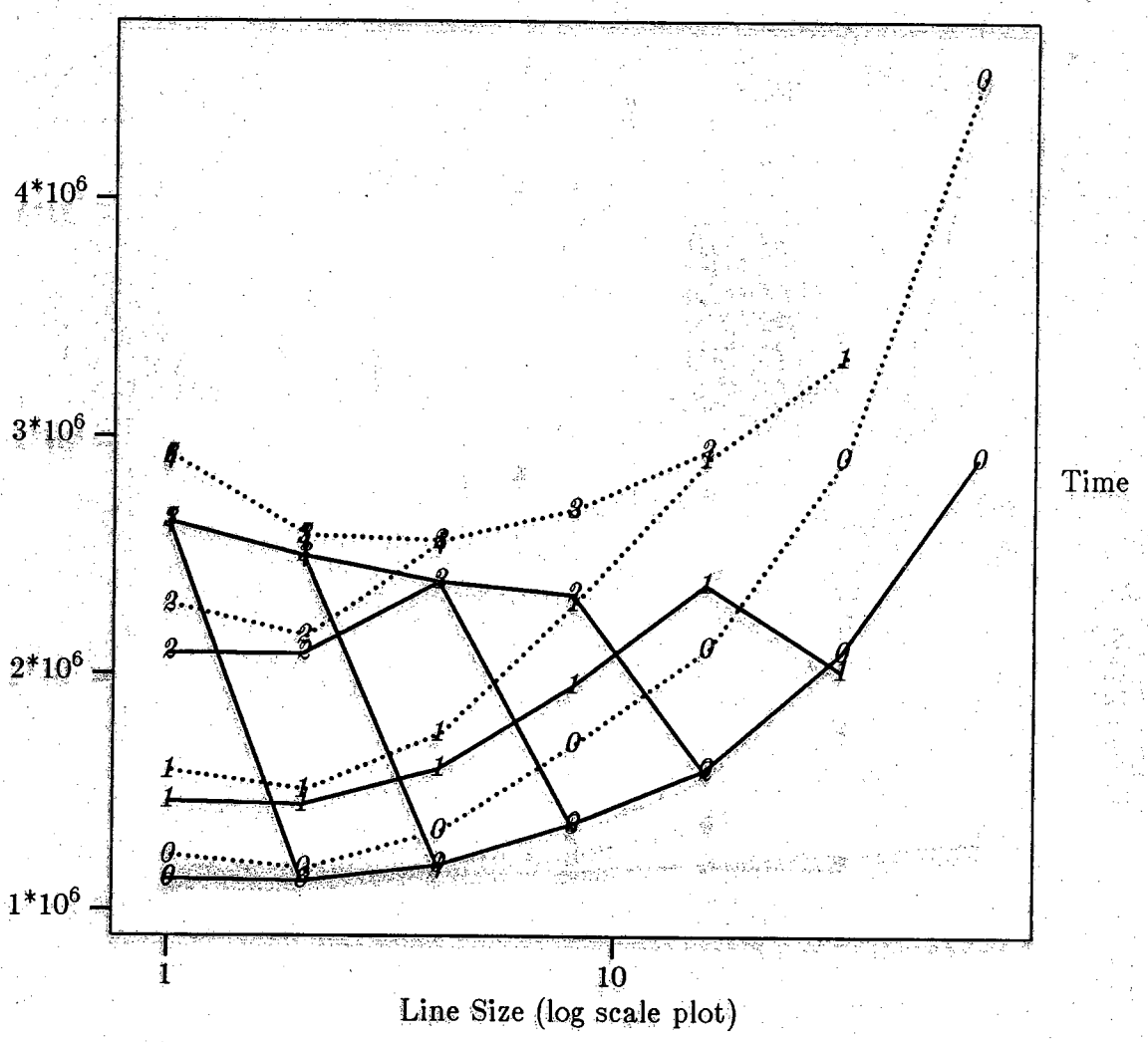


Figure 6.15: Reference Time of MAST Model and LRU Using Intmm, Instruction Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

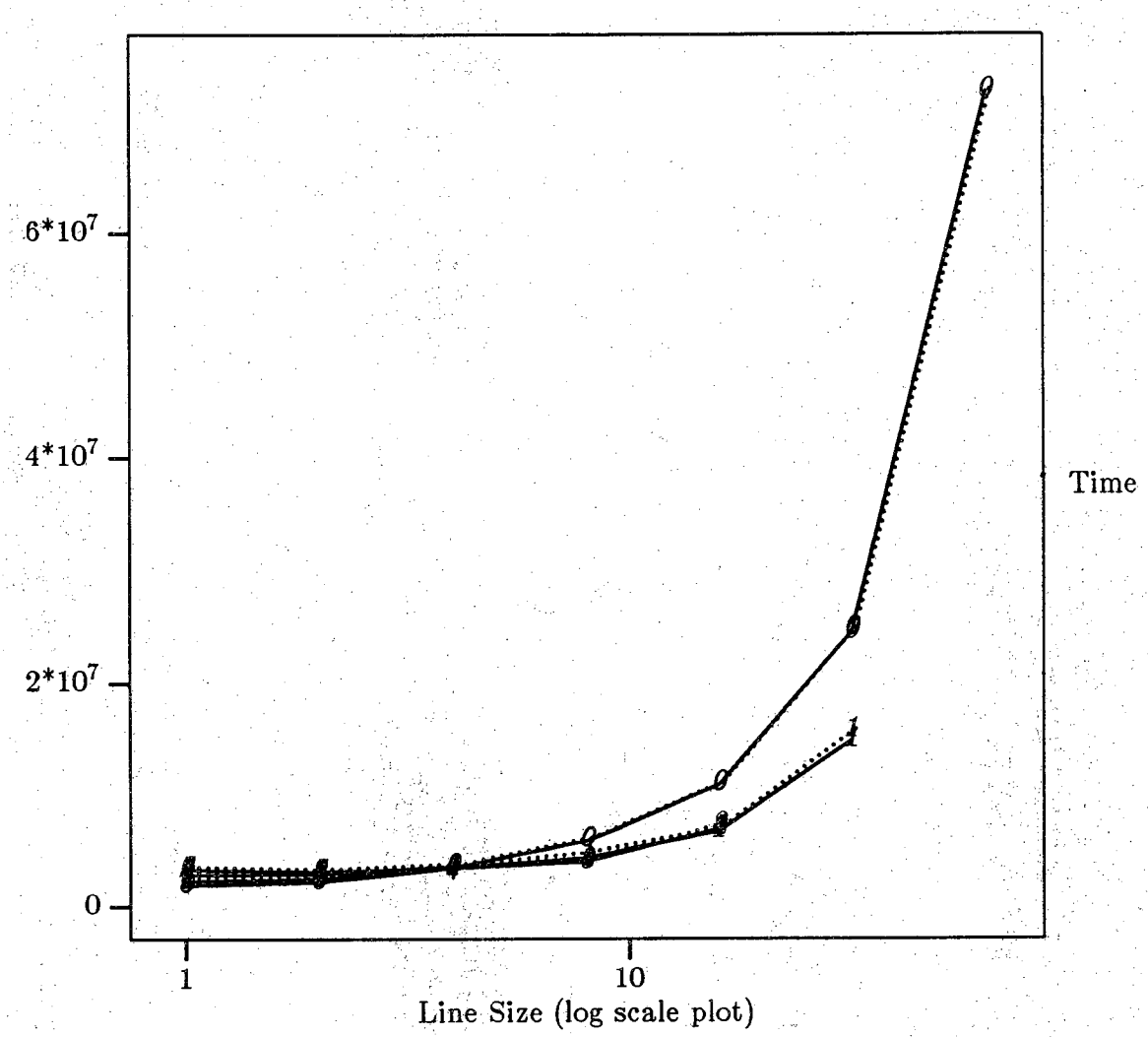


Figure 6.16: Execution Time of Liveness Model and LRU Using Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

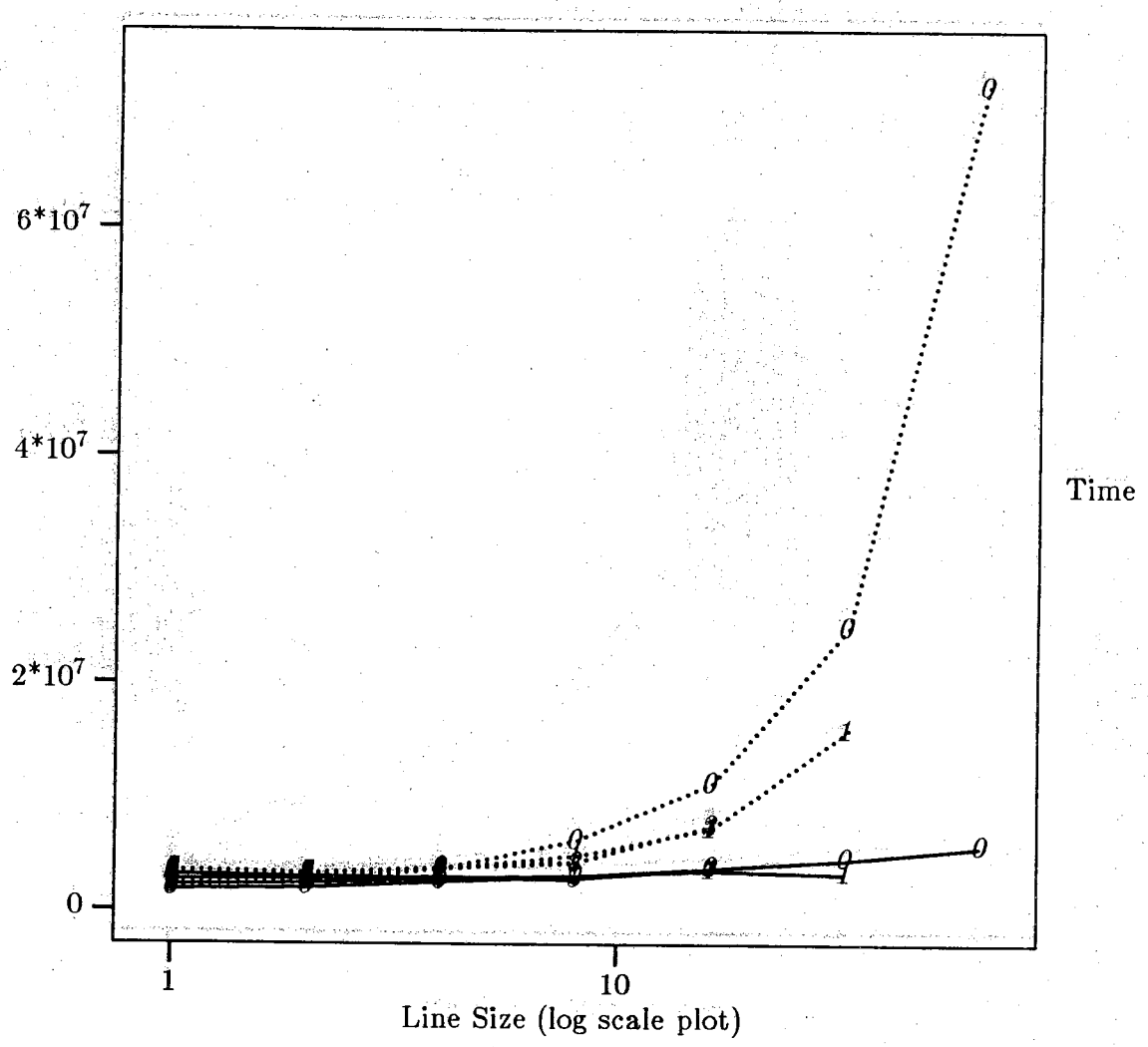


Figure 6.17: Execution Time of Bypass Model and LRU Using Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

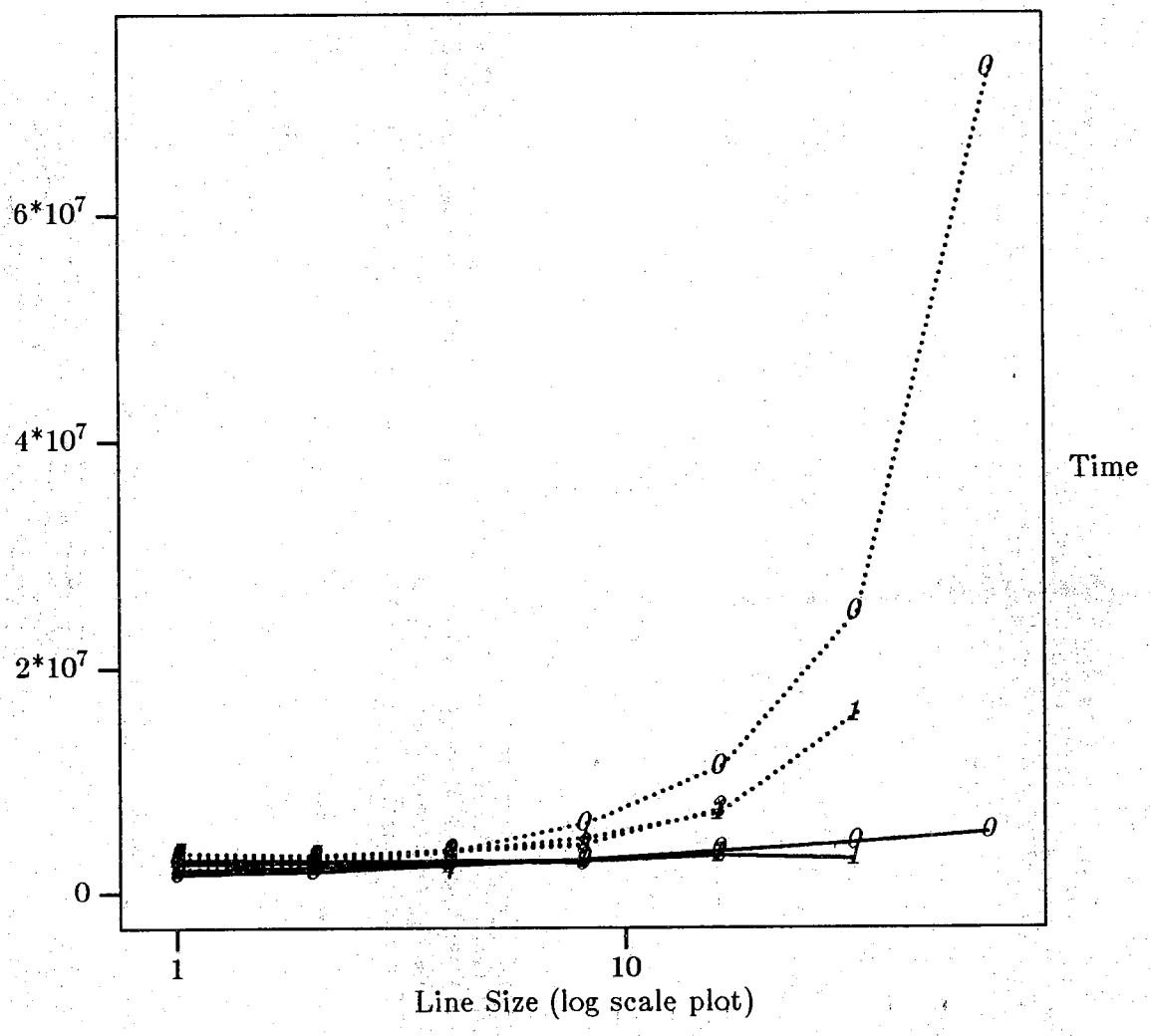


Figure 6.18: Execution Time of MAST Model and LRU Using Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

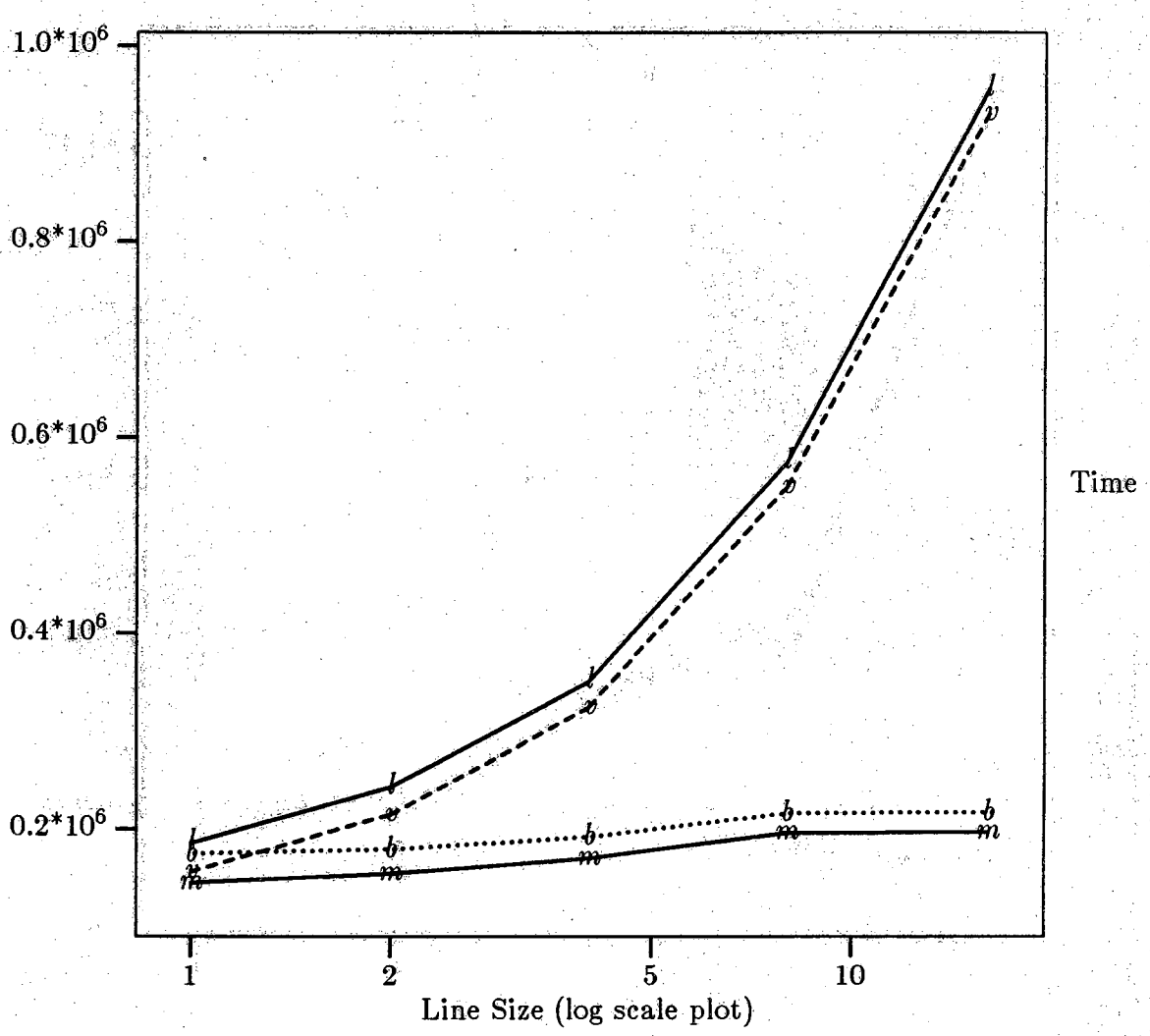


Figure 6.19: Execution Time Using Four Models, Intmm, Data Cache of Size 64, Set Size of 4, Access Ratio of 10, and Saving Factor of 25%

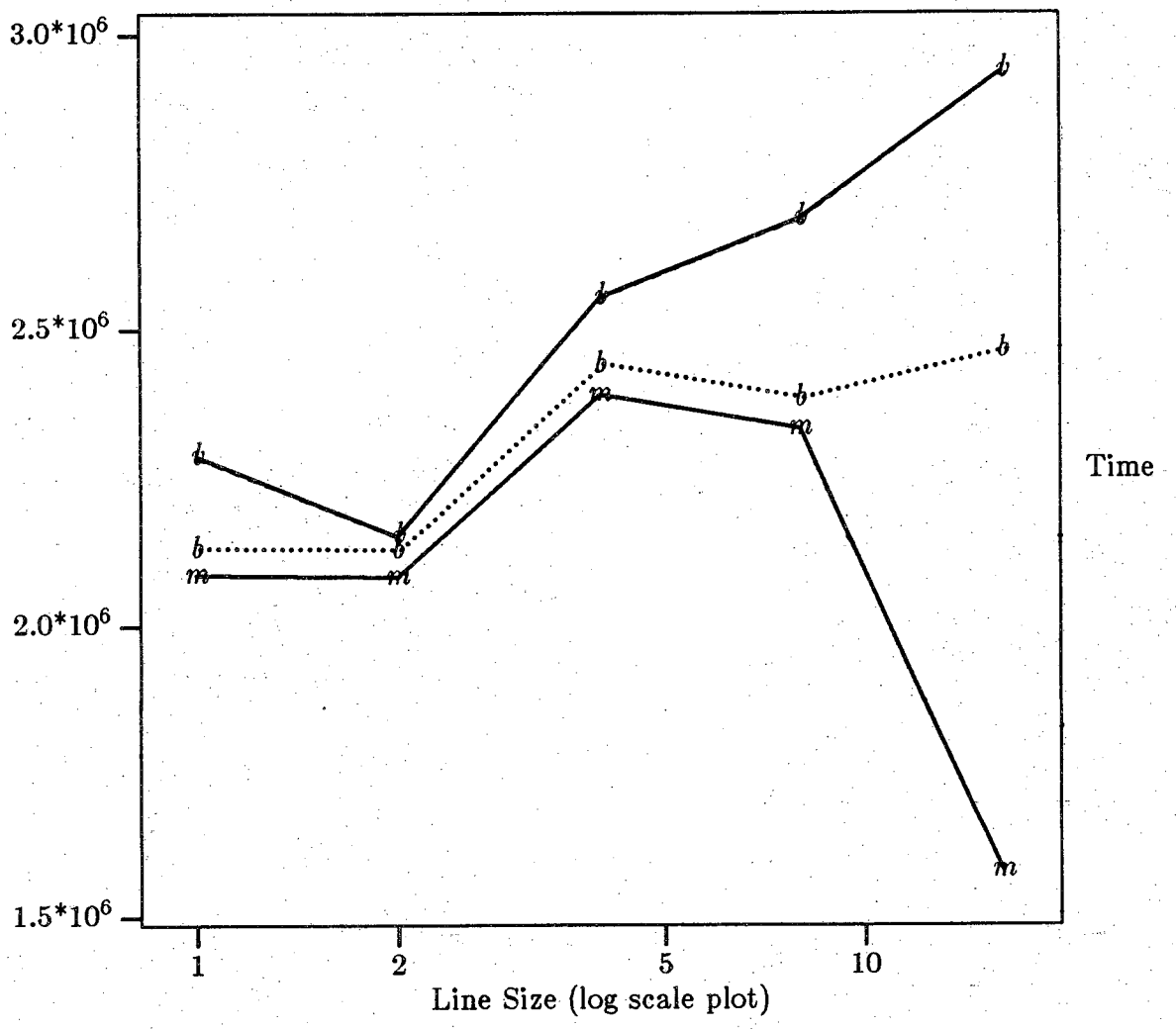


Figure 6.20: Execution Time Using Four Models, Intmm, Instruction Cache of Size 64, Set Size of 4, Access Ratio of 10, and Saving Factor of 25%

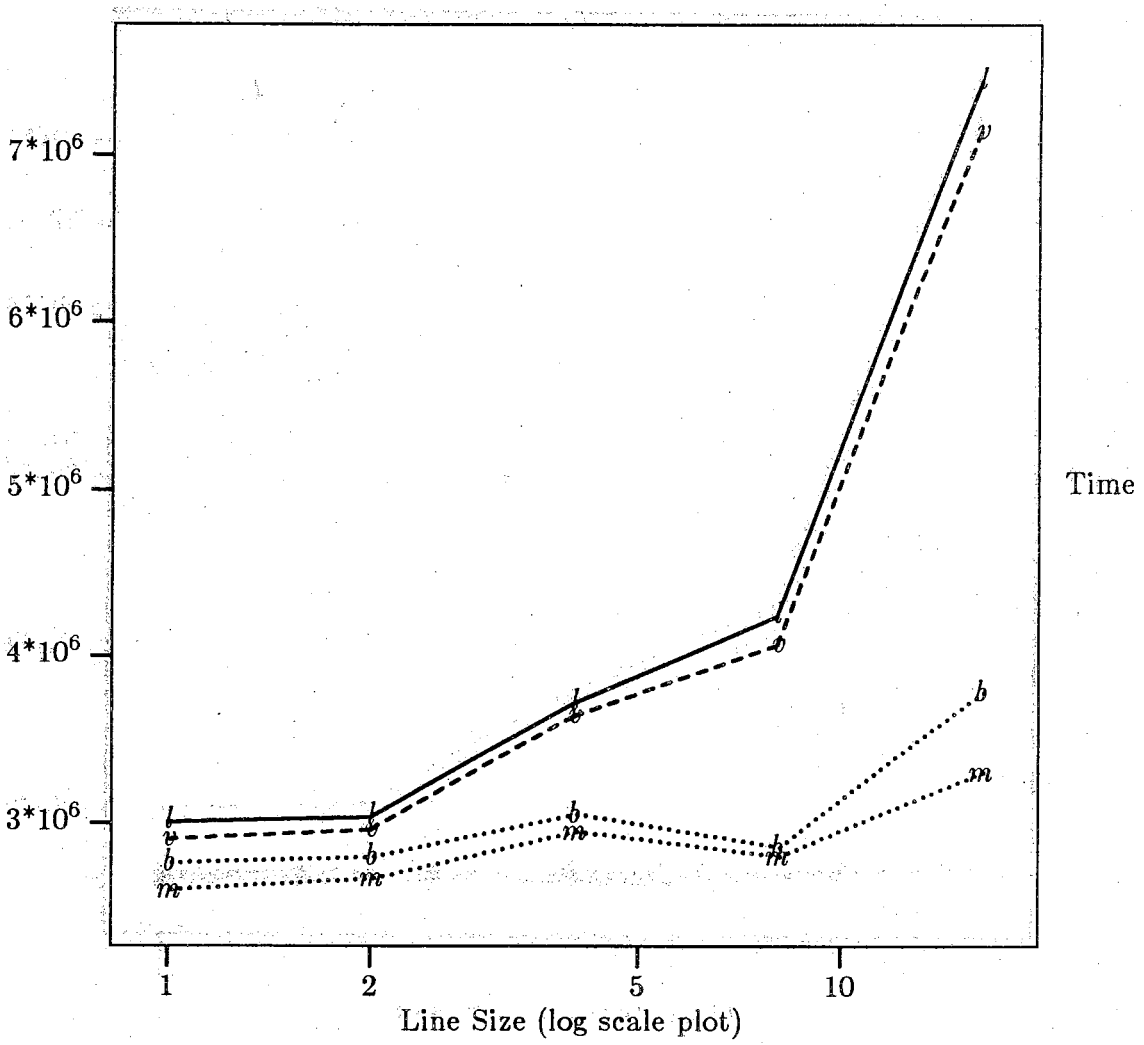


Figure 6.21: Execution Time Using Four Models, Intmm, Mixed Cache of Size 64, Set Size of 4, Access Ratio of 10, and Saving Factor of 25%

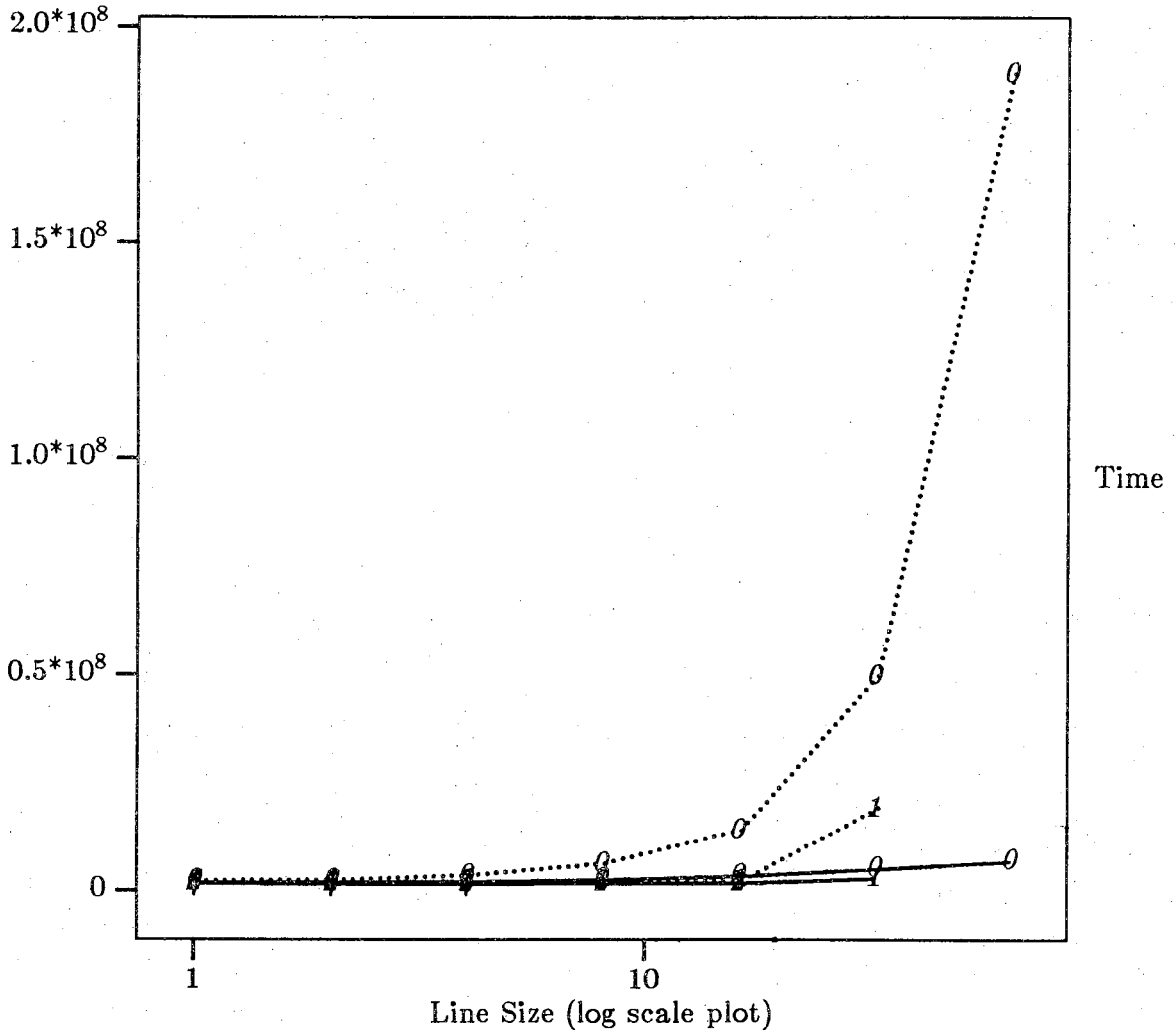


Figure 6.22: Reference Time of Bypass Model and LRU Using Bubble, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

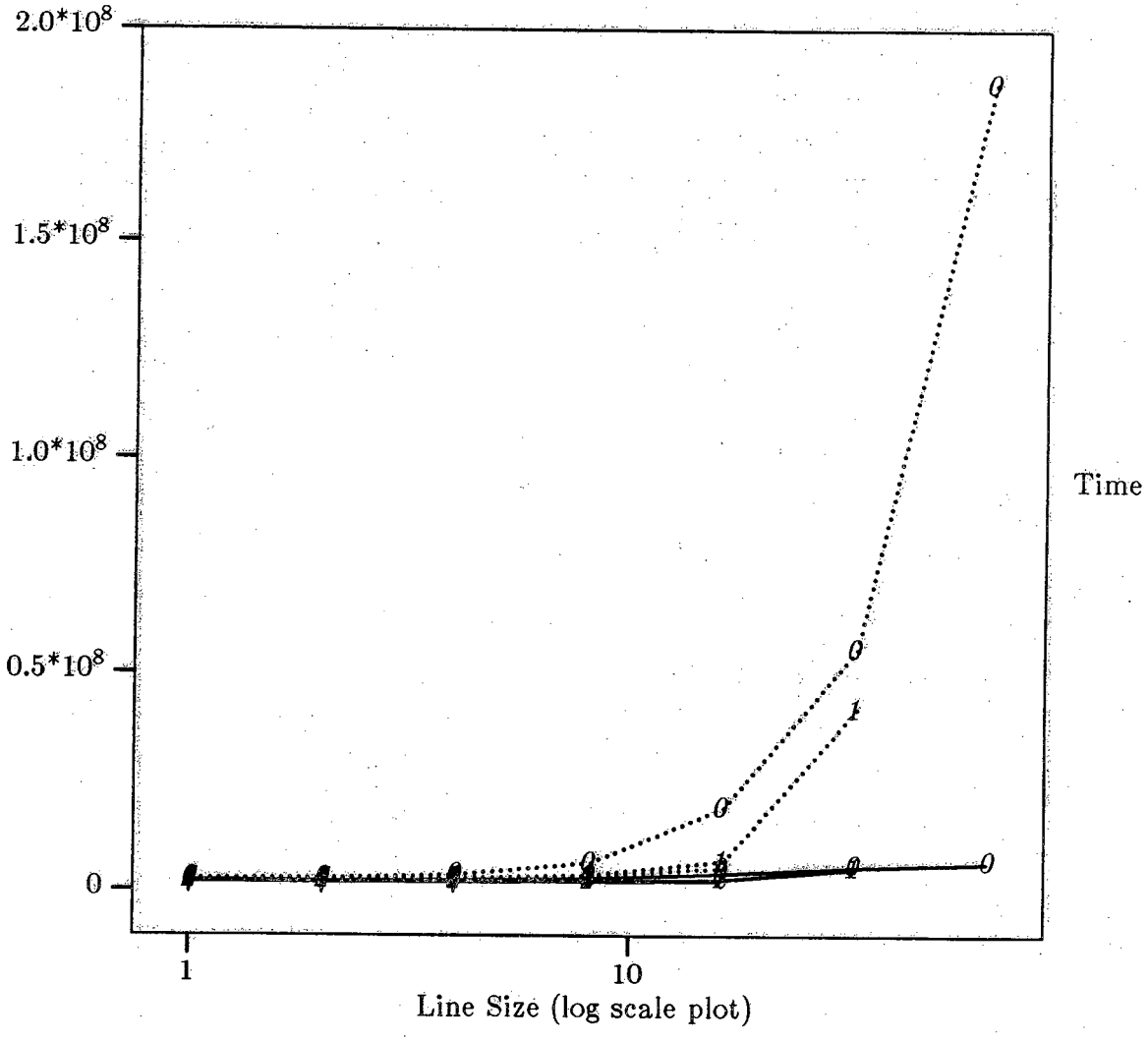


Figure 6.23: Reference Time of Bypass Model and LRU Using Puzzle, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

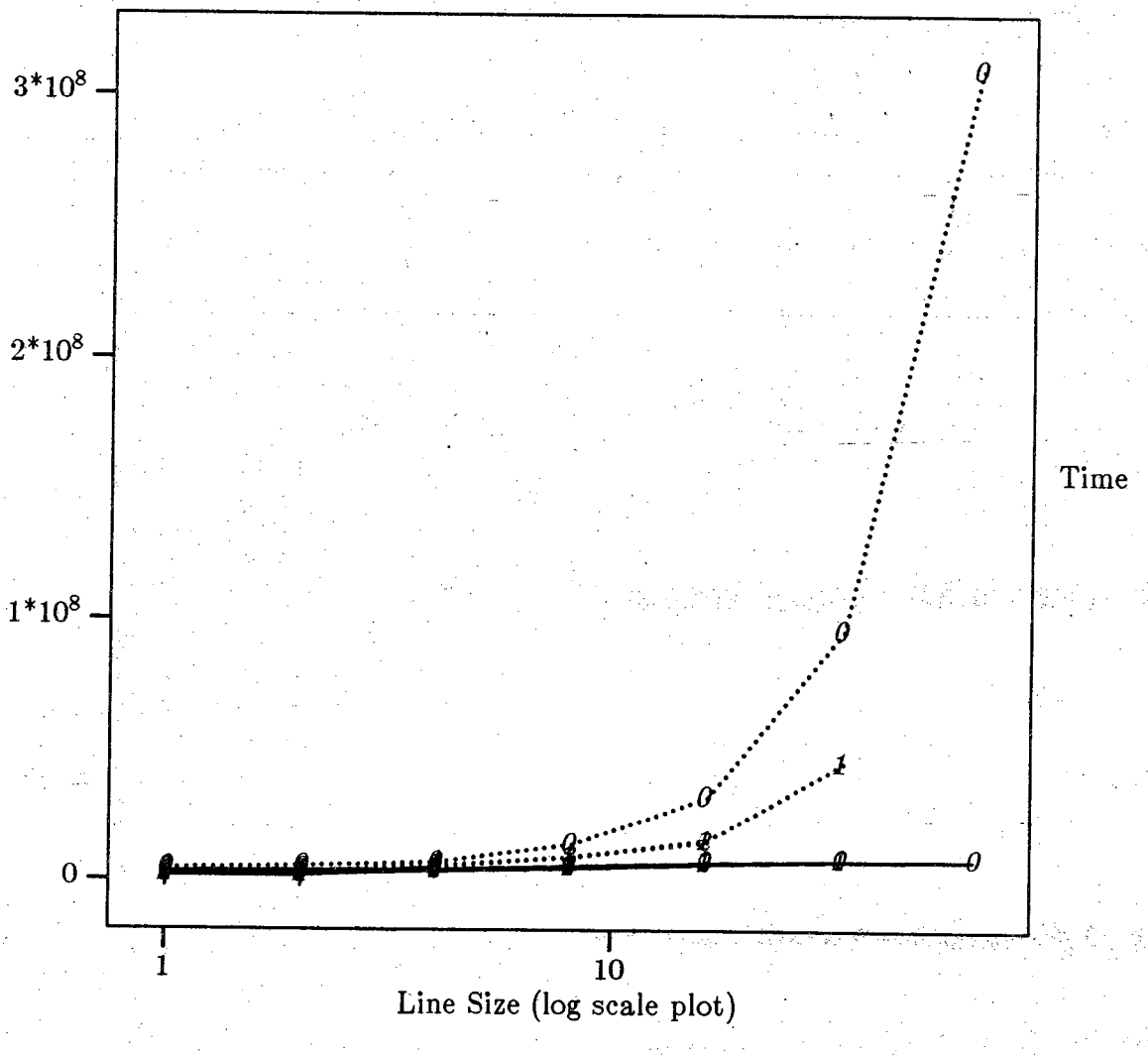


Figure 6.24: Reference Time of Bypass Model and LRU Using Towers, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

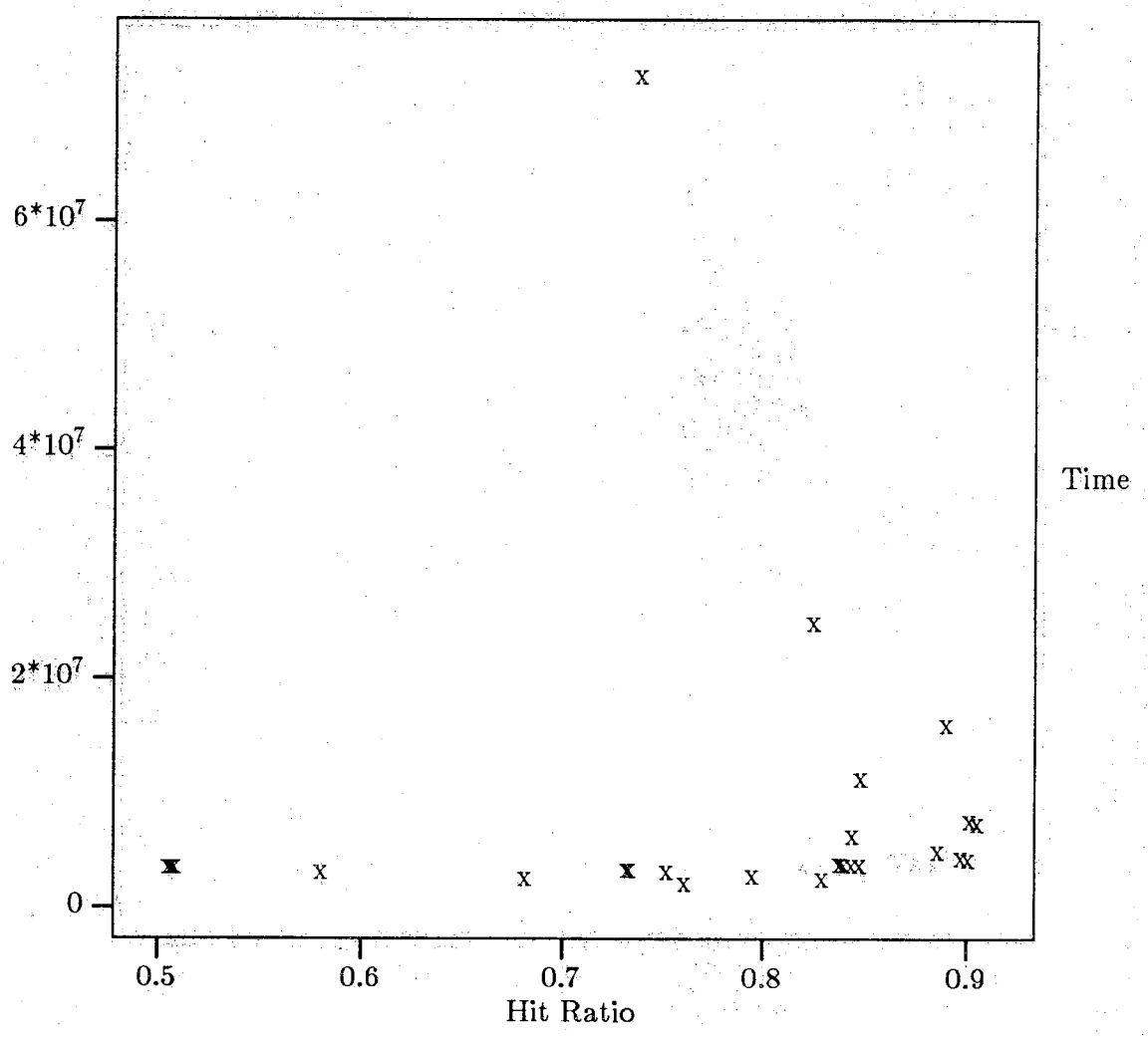


Figure 6.25: Reference Time vs Cache Hit Ratio Using LRU, Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

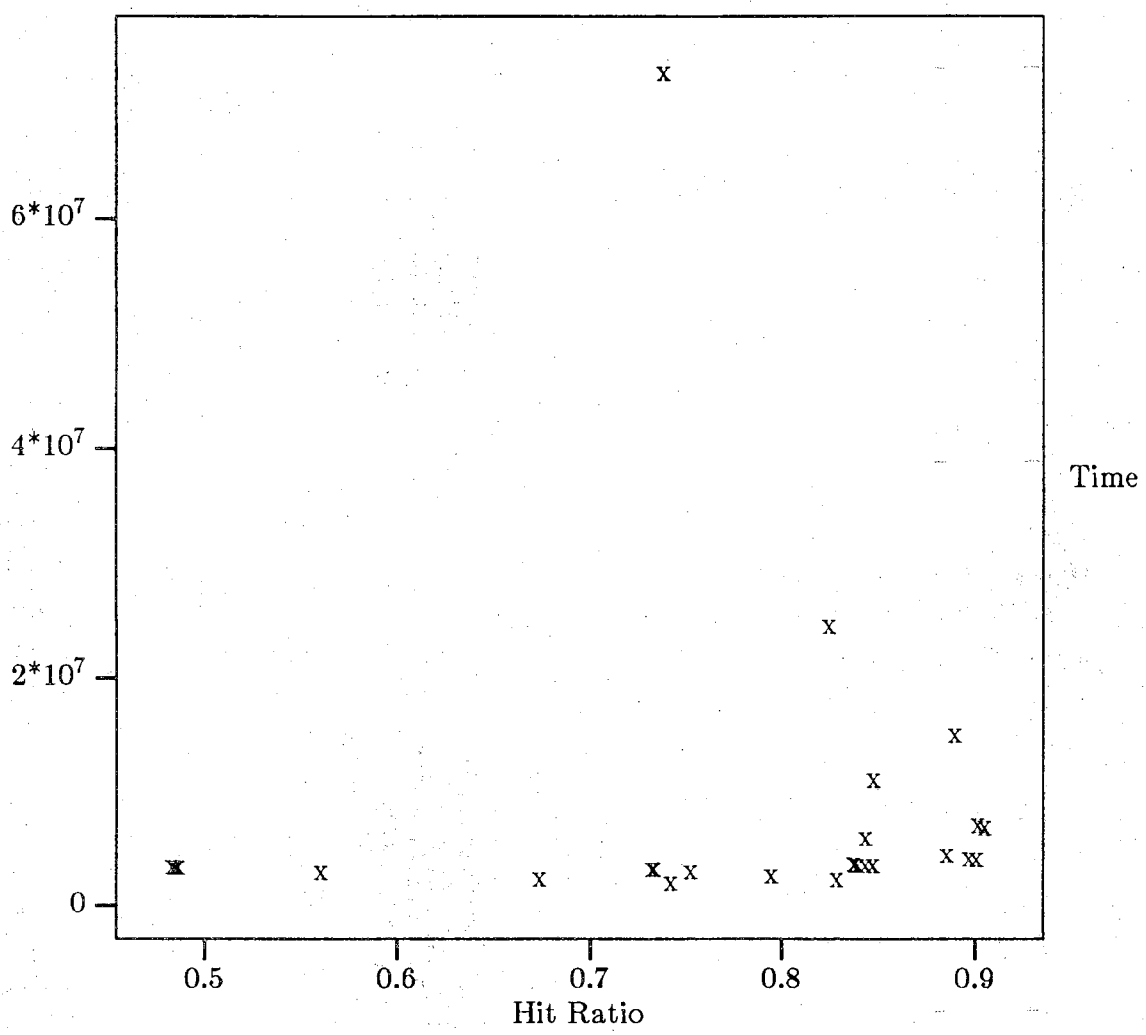


Figure 6.26: Reference Time vs Cache Hit Ratio Using Liveness Model, Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

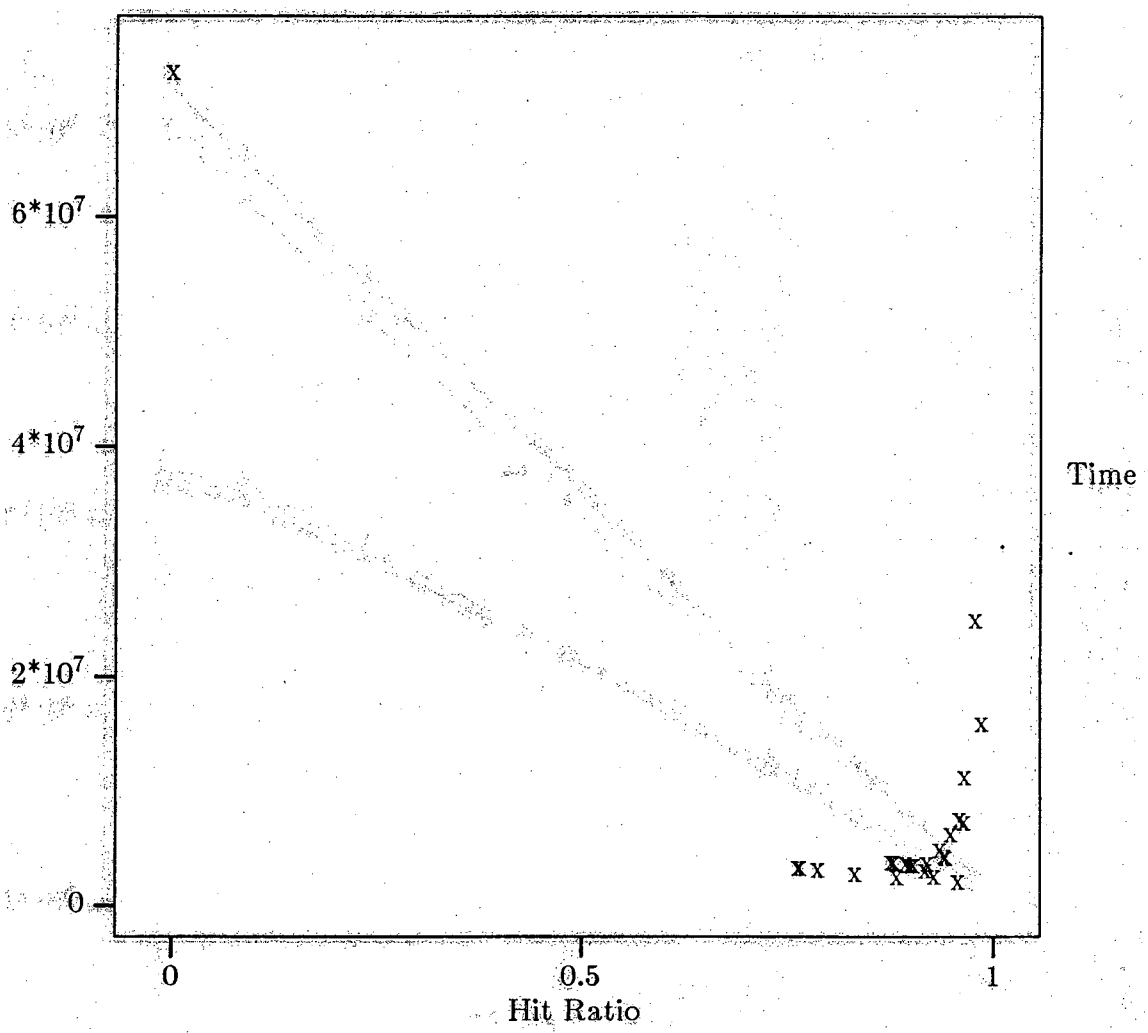


Figure 6.27: Reference Time vs Cache Hit Ratio Using Bypass Model, Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

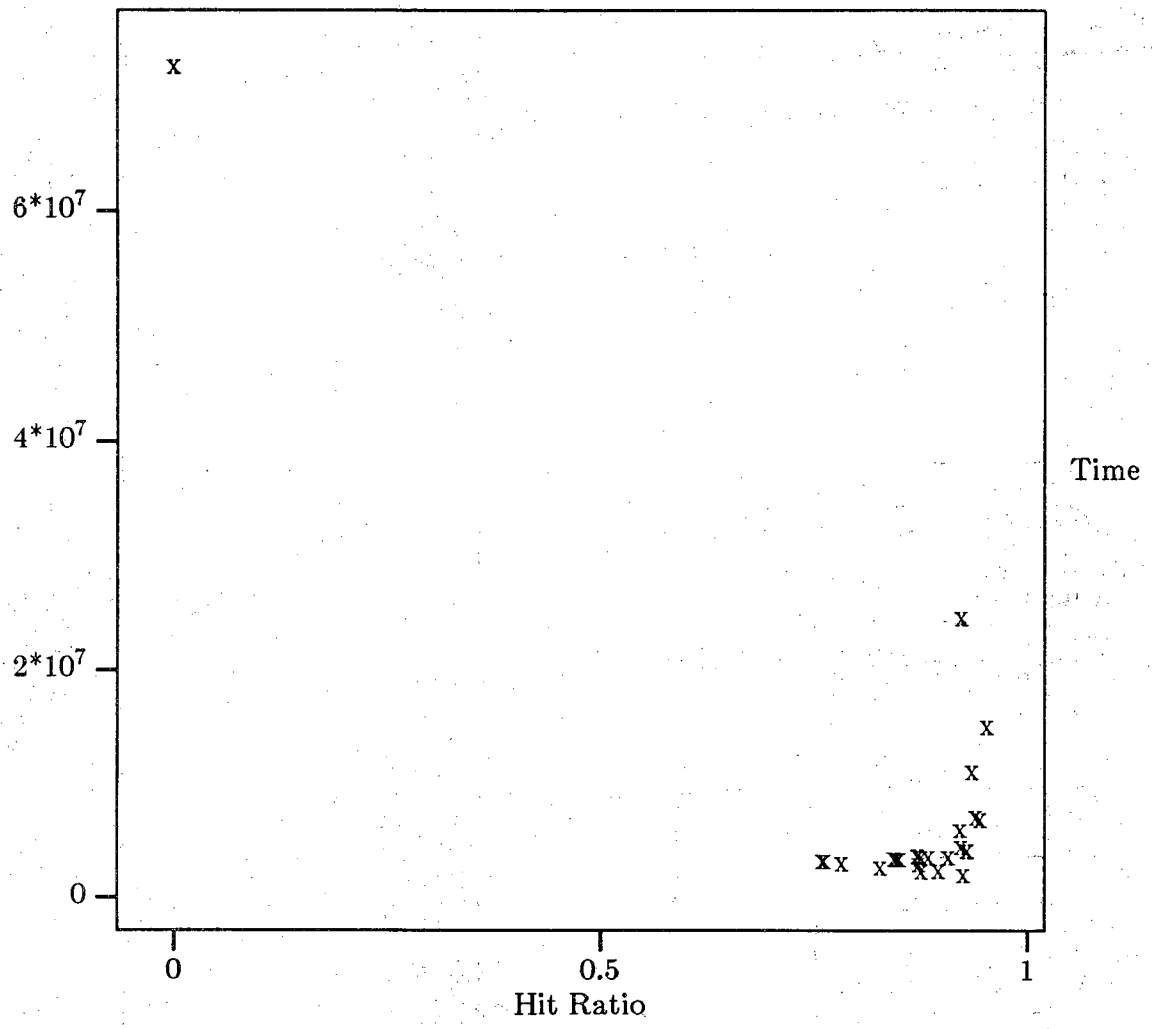


Figure 6.28: Reference Time vs Cache Hit Ratio Using MAST Model, Intmm, Mixed Cache of Size 64, Access Ratio of 10, and Saving Factor of 25%

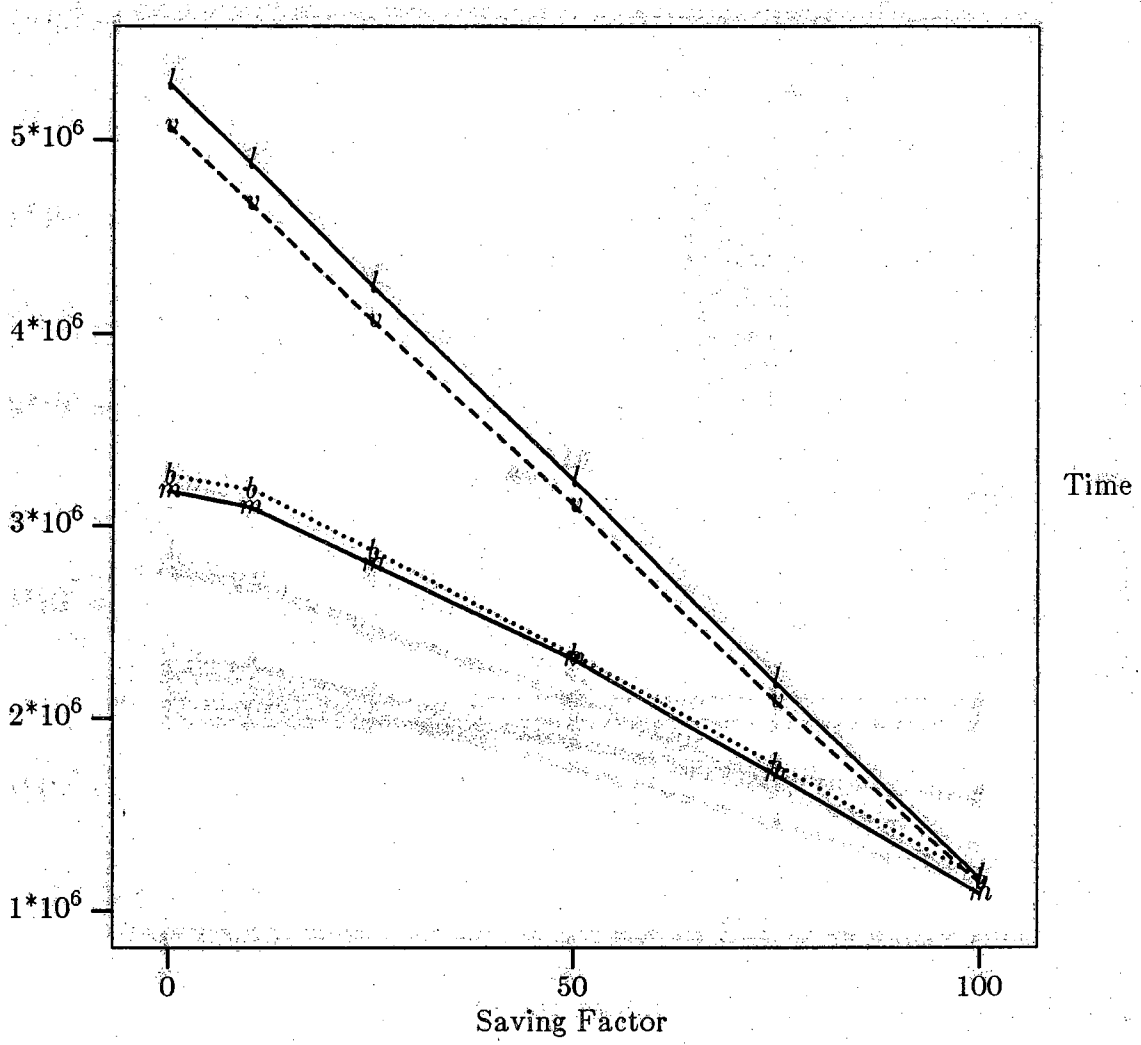


Figure 6.29: Reference Time vs Saving Factor Using Intmm, Mixed Cache of Size 64, Set Size 4, Line Size 8, and Access Ratio of 10

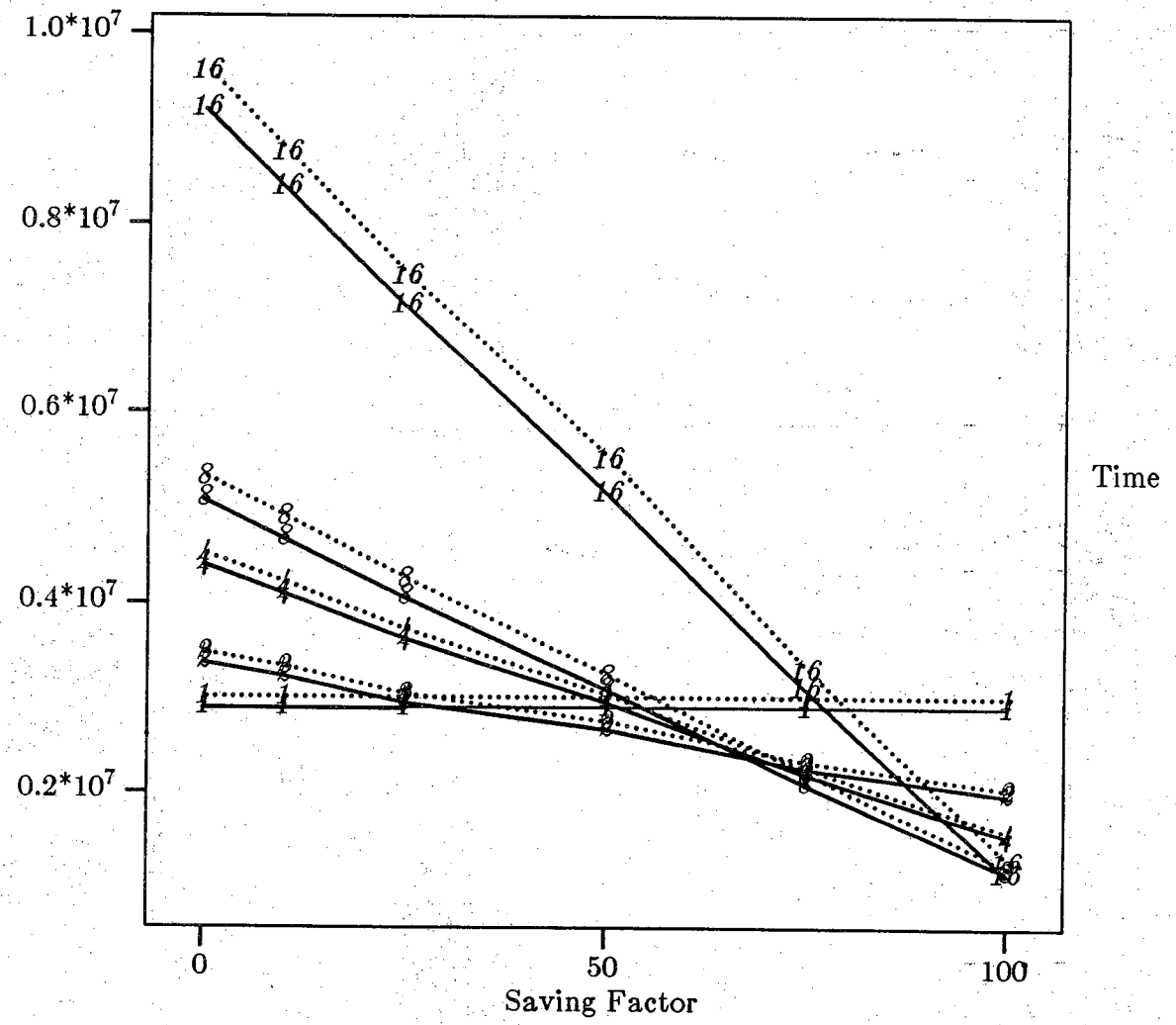


Figure 6.30: Reference Time vs Saving Factor Using Liveness Model and LRU, Intmm, Mixed Cache of Size 64, Set Size 4, and Access Ratio of 10

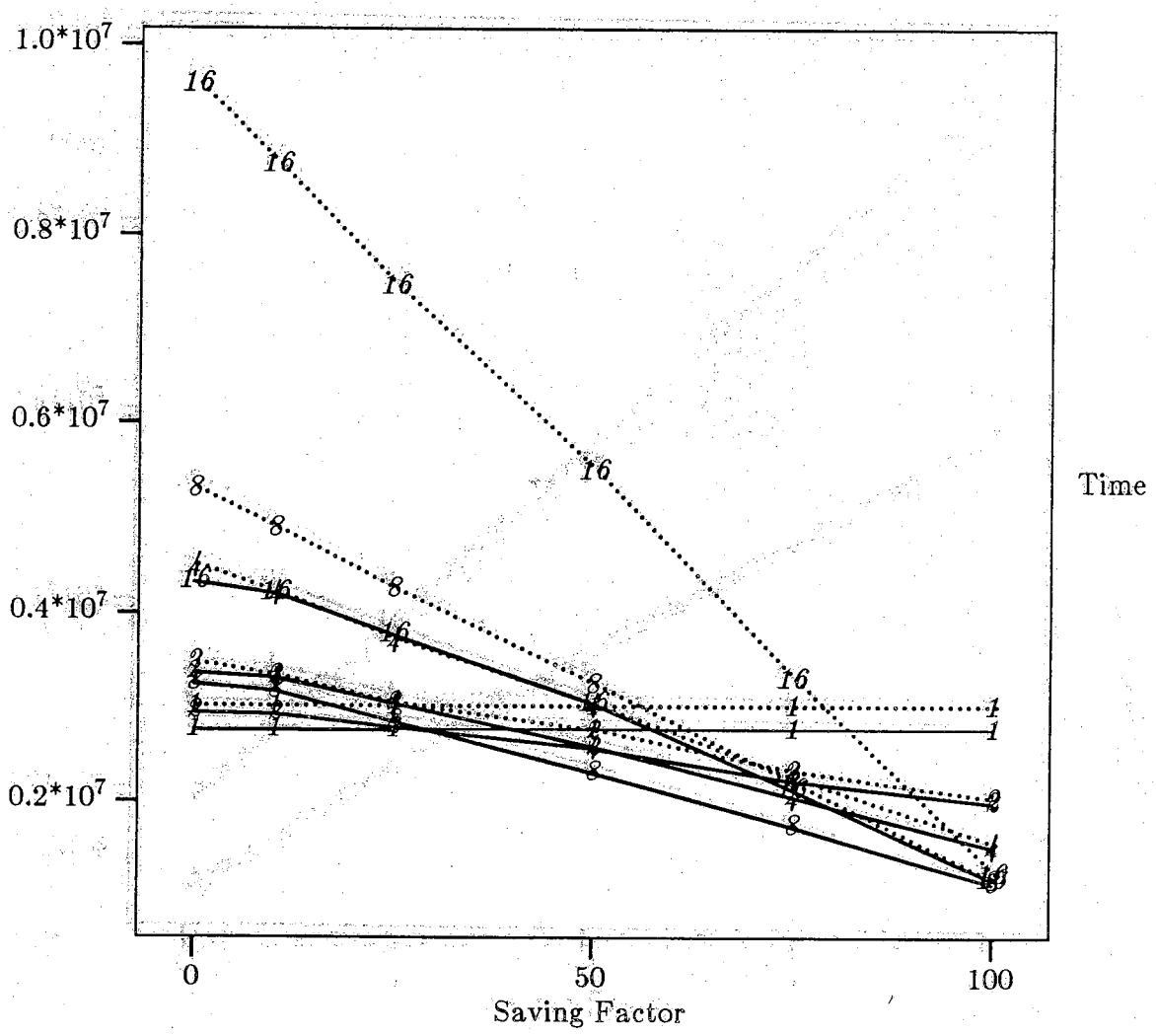


Figure 6.31: Reference Time vs Saving Factor Using Bypass Model and LRU, Intmm, Mixed Cache of Size 64, Set Size 4, and Access Ratio of 10

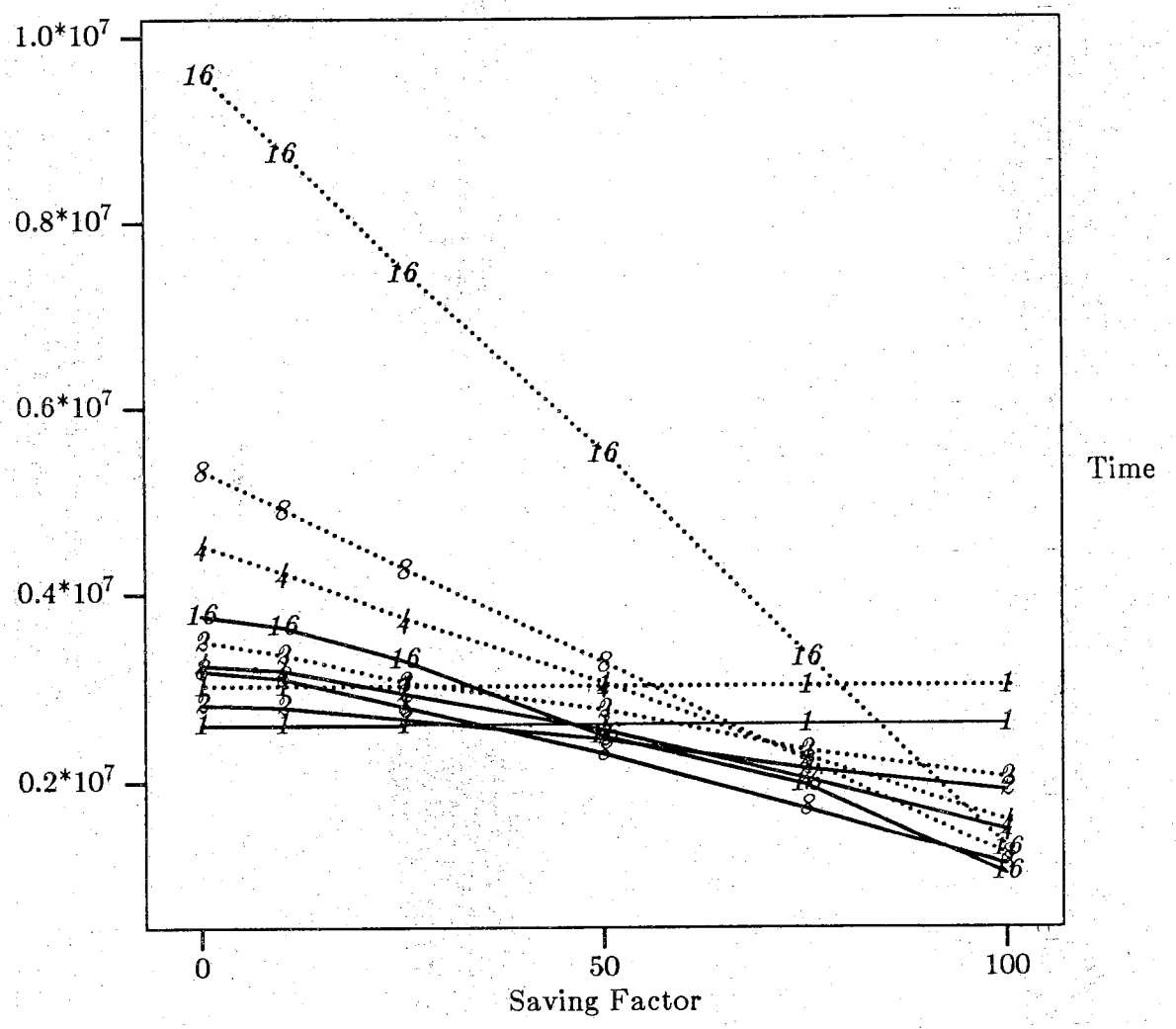


Figure 6.32: Reference Time vs Saving Factor Using MAST Model and LRU, Intmm, Mixed Cache of Size 64, Set Size 4, and Access Ratio of 10

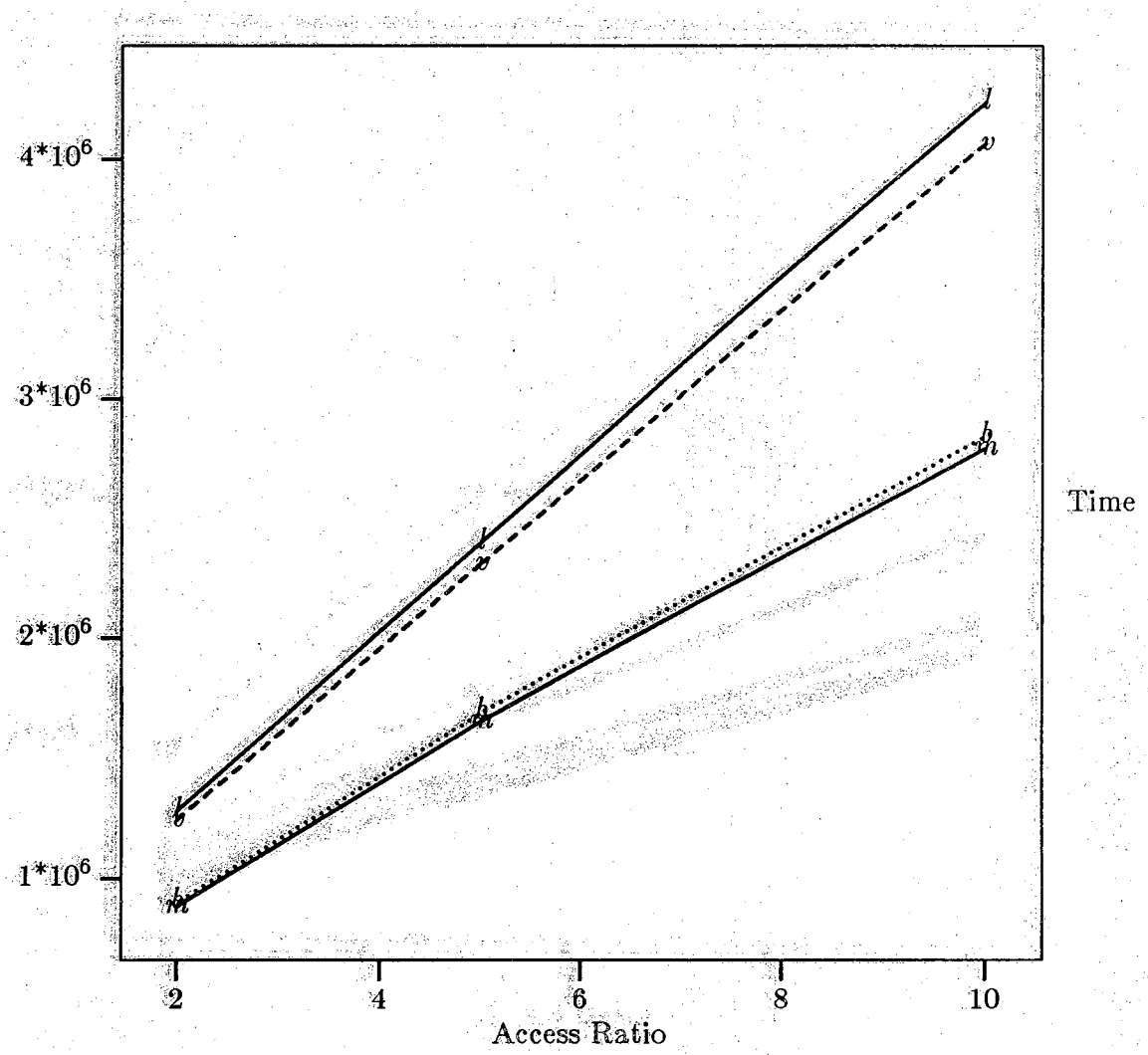


Figure 6.33: Reference Time vs Acess Ratio Using Intmm, Mixed Cache of Size 64, Set Size 4, Line Size 8, and Saving Factor of 25%

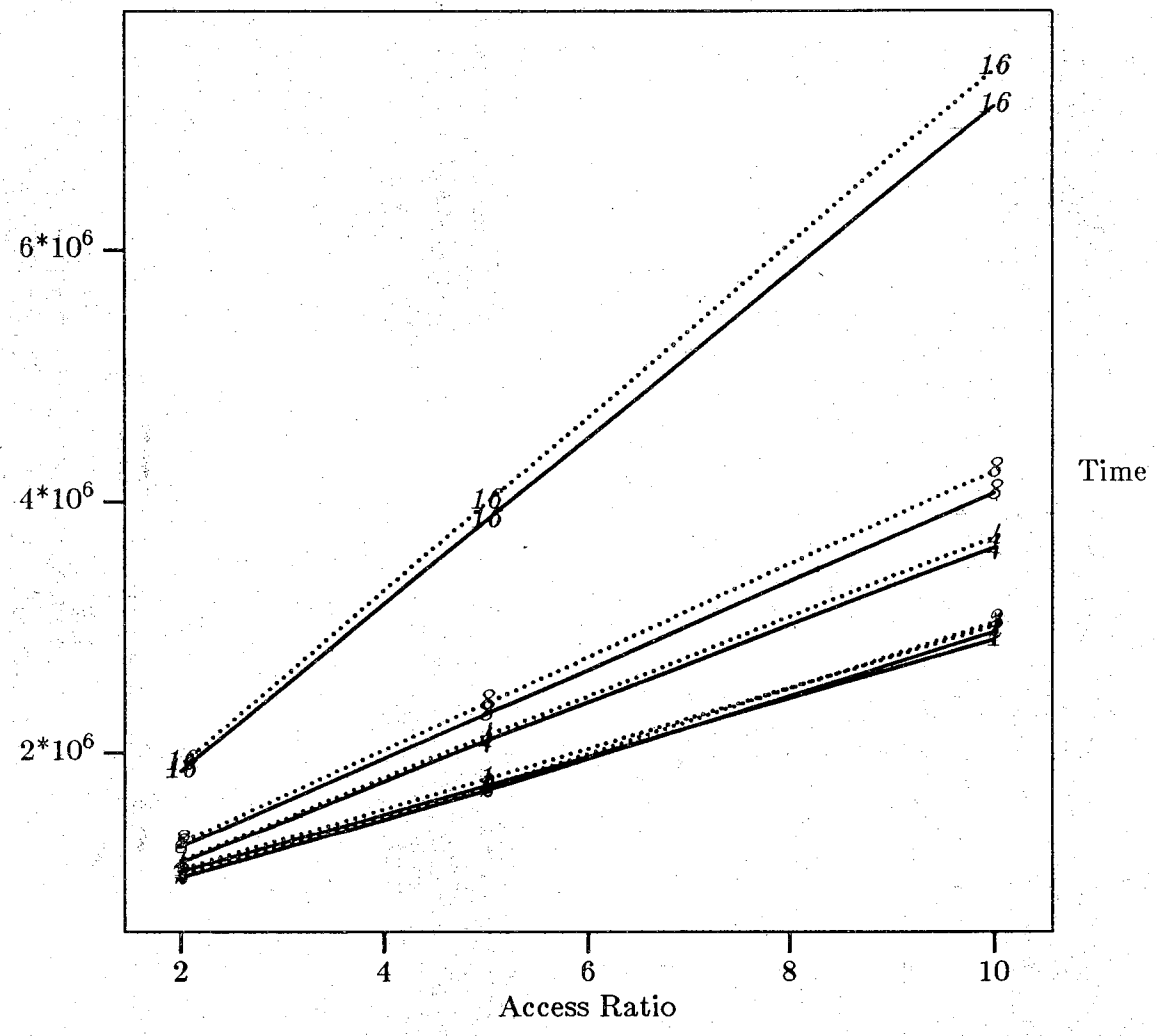


Figure 6.34: Reference Time vs Acess Ratio Using Liveness Model and LRU, Intmm, Mixed Cache of Size 64, Set Size 4, and Saving Factor of 25%

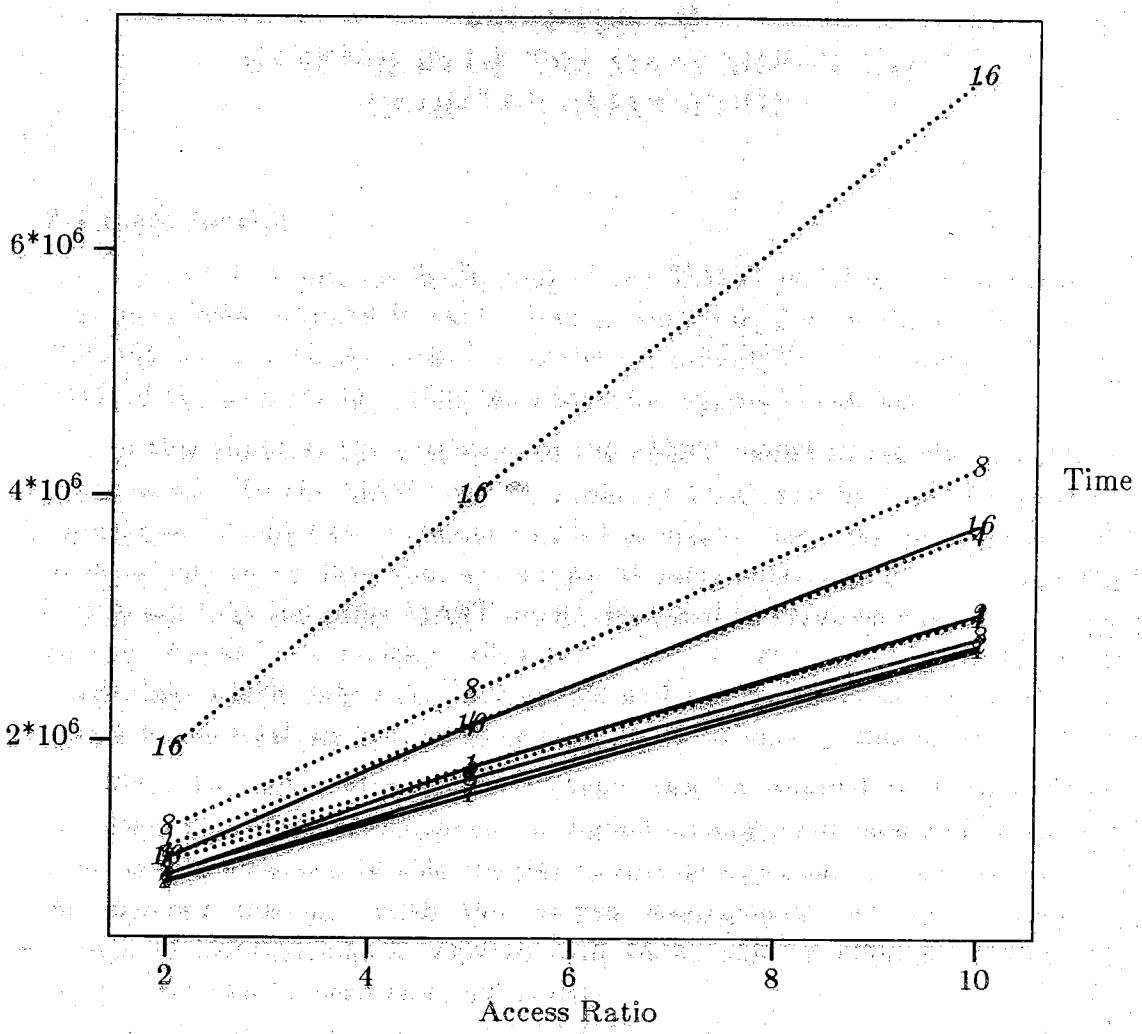


Figure 6.35: Reference Time vs Acess Ratio Using Bypass Model and LRU, Intmm, Mixed Cache of Size 64, Set Size 4, and Saving Factor of 25%

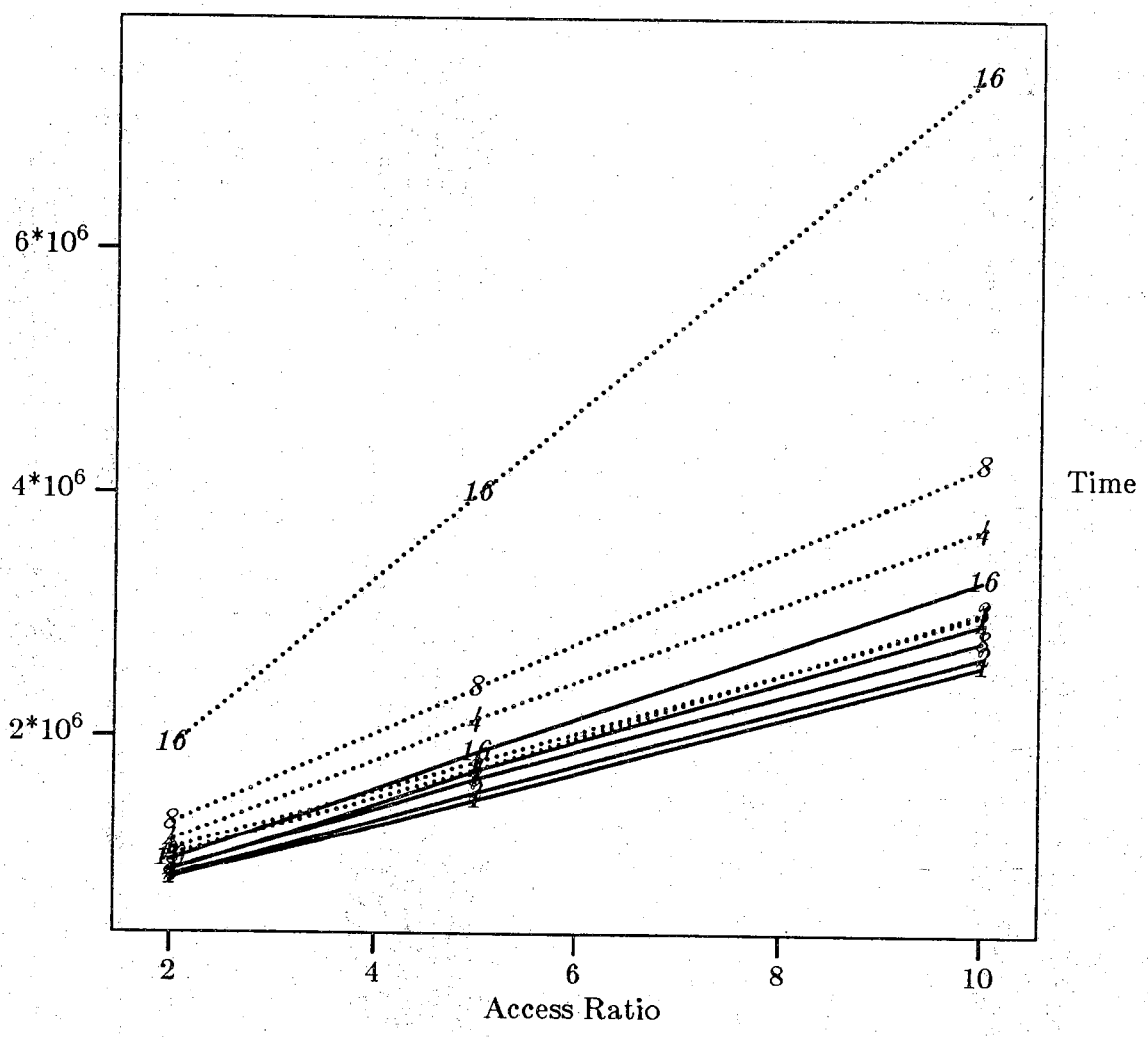


Figure 6.36: Reference Time vs Acess Ratio Using MAST Model and LRU, Intmm, Mixed Cache of Size 64, Set Size 4, and Saving Factor of 25%

CHAPTER VII
EXTENSION OF THE MAST MODEL TO REGISTER ALLOCATION

7.1 Introduction

Up to this point, all discussions of the MAST model and its generative uses have been confined to cache design. However, due to the similarities of different memory hierarchical levels, the same model can be applied to other levels of the memory hierarchy with little (or no) modifications.

In this chapter, the extension of the MAST model to register allocation is discussed. To the MAST model, a register array can be viewed as a fully associative "cache-like" structure with line size of one. Its main purpose is to store only those data that are known at compile-time to be unambiguous. It is found that the same MAST model discussed in previous chapters can be directly applied to register allocation with no changes. In fact, those constraints which only exist in registers and not in cache only simplify the MAST model analysis instead of causing any problems or difficulties.

Since the same management scheme can be applied to both register allocation and cache management, a unified management model of these two levels is also proposed in this chapter to further enhance the performance of the memory system. With this unified management scheme, redundant storage of information in registers and cache can be eliminated and the cache space can be used more efficiently.

The outline of this chapter is as follows. In Section 7.2, a brief survey of existing register allocation techniques is given. A new heuristic for register allocation using graph coloring, known as the **random walk coloring**, is proposed in Section 7.3 as a simple technique to improve register allocation compared to standard graph coloring and spilling [ChA81] [Cha82]. In Section 7.4, the similarities between the registers and cache are first identified. Then, extension of the MAST model described in Chapter 3 and 4 to perform register allocation is presented. In Section 7.5, a **unified registers/cache management model** is proposed to improve the efficiency

of the memory hierarchy. This chapter finally concludes in Section 7.6.

7.2 Existing Register Allocation Techniques

Since the first FORTRAN compiler for the IBM 704 and 709, the issue of mapping values used in a program to a finite number of machine registers has been considered as one of the classic problems in compiler optimizations [AhS86]. It is generally agreed that the quality of register allocation is a key element in the efficiency of program execution [Bac57]. Compiler writers always attempt to design techniques which will place as many unambiguous data references (uses and definitions) as possible into registers.

Various register allocation schemes based on different criteria have been proposed since the late 1960s. These schemes can be divided into three basic approaches based on the criteria used to allocate values into registers:

- register allocation by machine level state analysis [HoK66], [Luc67], [Ken71], [Ken72], [Agr79],
- register allocation via usage counts [Fre74],
- register allocation and spilling via graph coloring [ChA81] [Cha82], and
- register allocation by priority-based coloring [Har75] [Har83] [ChH84] [Cho83].

A brief survey of these register allocation schemes is given in the next section.

7.2.1 Register Allocation by Machine Level State Analysis

In the mid 1960s, compiler writers started to investigate various code optimizations by the compiler. One of the most important code optimization problems that they studied was to find an efficient scheme to map indices used in a program into the three index registers available in the IBM 709 and 7090 machine.

In 1966, Horwitz, Karp, Miller and Winograd proposed an algorithm for allocating index registers [HoK66]. In their algorithm, all possible allocations of index registers for a given straight line reference sequence are considered. A machine level state transition graph is formed where a node in the graph is a feasible register configuration at some stage in a program and an edge is a feasible transition from one configuration to the next.

At each step *i* in a program, a node at step *i-1* in the graph is associated with every register configuration at step *i*, which might occur at this step. The cost of an edge joining two register configurations of successive steps is defined as the transition cost from one register configuration to the other. The problem of index register allocation is then reduced to the problem of finding a "shortest path" (or the lowest cost) through a directed graph, starting from a given initial register configuration to some register configuration associated with the last step of a program. That is, the optimal register allocation for a given straight line reference sequence is the path through the graph which has the minimum sum of the costs of those edges along that path. Since the number of possible register configurations in the graph is an exponential function of the number of distinct references in a program, various heuristics were proposed to minimize the computational complexity of the analysis.

In the early 1970s, Kennedy [Ken71] [Ken72] extended this index register allocation technique to general register allocation problem with simple loops and boundary conditions. However, this register allocation technique was considered infeasible and faded away after 1972 mainly because:

- the computational complexity of the analysis was considered too high to be practical — the concept of cut point and reference live range was not available in their analysis,
- the technique was only proposed for a straight line reference sequence and no consideration of other program structures was made in the analysis, and
- program data flow analysis in the 1970s was immature.

In fact, this register allocation technique has never been implemented.

Despite all the above problems, this approach actually points out a very important fact: the cost of each register configuration transition should be considered for any optimal register allocation scheme. Since this approach was initially proposed for index register allocation, the option of register bypass was not considered.

After the index register allocation model, register allocation was generally considered too complex for optimal algorithms to be developed and applied. A lot of effort was then spent in transforming the register allocation problem into some well-known problems such as graph coloring

[ChA81] [Cha82], and usage counts [Fre74]. A long sequence of ad-hoc techniques have also been developed and refined to improve the quality of the register allocation. Not surprisingly, improvements in the register allocation schemes came through added complexity in handling extra bias factors, rather than simplifying and generalizing the techniques.

7.2.2 Register Allocation Via Usage Counts

In 1974, Freiburghouse [Fre74] introduced the notion of usage counts and used the usage count as the basis for a register allocation scheme. Instead of employing the execution-time usage count, he defined the **usage count** of a value as the number of distinct references made to that value within the program text (at compile time).

He further divided the register allocation problem into two sub-problems: (1) the **value retention** problem, and (2) the **register demand** problem. The value retention problem is that the code generator needs to insure that a value is no longer be held in a register when it is no longer needed. The register demand problem is that the code generator needs to choose which register to overwrite when a value must be placed in a register and no registers are available.

To solve these problems, Freiburghouse's algorithm records which value is in each register and the corresponding usage counts. When the compiler generates code referencing a value, the usage count of that value is decremented by one. A usage count of zero means that the register is "free," hence, it can be reused for some other value. When a value must be placed in a register, but no register is free, then a register is made free by **spilling**. To spill, the register with the lowest usage count is chosen, and the value it holds is stored into memory *iff* the value is dirty (i.e., the value in the register does not match the value of that variable in memory).

Since the usage count scheme described above is easy to implement, and since it performs reasonably well most of the time, many compilers employ this technique. Theoretically, however, the usage count scheme only operates correctly in that, if loops and conditionals are ignored, the freeing of a register when the reference count becomes zero is correct. The reasonable performance of the usage count scheme in general is by serendipity rather than by design.

Although the usage count scheme assumes that every value that could be placed in a register should be placed in a register, this is not generally true — even if there is an infinite supply of free registers. This is because an overhead of one register load instruction is required to place a datum into register. If the reference frequency of a datum is low, it might sometimes be beneficial to use instructions with one or more direct memory reference operations, provided that this option is allowed by the machine. Table 7.1 shows the four possible coding of an operation and their number of registers and instructions required.

Additional benefits can further be obtained if there are not sufficient registers available because the register load instruction might cause a register spill.

Table 7.1: Possible Coding for D = S1 OP S2

Coding	No. of Reg. Used	No. of Instr.	No. of Mem. Operand
Load R0, Addr(S1) Ldoa R1, Addr(S2) OP R2, R0, R1 Store addr(D), R2	3	4	0
Load R0, Addr(S1) OP R1, R0, Addr(S2) Store Addr(D), R1	2	3	1
OP R1, Addr(S1), Addr(S2) Store Addr(D), R1	1	2	2
OP addr(D), Addr(S1), Addr(S2)	0	1	3

The other problem with register allocation via usage counts is that if register spilling must occur, the register with the lowest usage count is selected. The fact that the usage count is lowest is, however, totally unrelated to how soon the next reference of that value will occur and is also unrelated to any value "lifetime packing",[1] both of which are vital in

[1] Lifetime packing refers to the idea that two values with non-overlapping lifetimes can share a single register. Therefore, under certain circumstances, the best value to

achieving optimal register allocation. For example, a value with the lowest usage count may be referenced next and a value with the highest usage count may be referenced only at the very end of the basic block to which it belongs. For example, given the reference sequence in Figure 7.1:

Ref. Sequence No.	0	1	2	3	4	5	6	7	8	9
Value	*C*	*D*	*E*	*E*	*D*	*E*	*C*	*C*	*C*	*C*

Figure 7.1: Value Reference Sequence

and assuming the machine provides two registers, the usage count register allocation scheme will retain the value *C* in a register during the execution of references 0 through 9 (because it has the highest usage count) whereas if value *C* were spilled in the reference operation 2 instead of value *D*, better performance would ensue.

To perform optimal register allocation, far more precise information about the order of references must be obtained and used.

7.2.3 Register Allocation & Spilling Via Graph Coloring

Another common approach to register allocation is to treat register allocation as a graph coloring problem. Despite its proposal in the early 1970s [Yer71] [Sch73] [AlC76], this technique was first implemented in 1980, when an experimental PL/I compiler for the IBM System/370 embodied this approach in its register allocation phase [ChA81]. In 1982, Chaitin proposed a complete register allocation and spilling algorithm based on graph coloring [Cha82].

Under this scheme, the intermediate code entering the register allocator is written assuming that there is an infinite number of fast temporary memory cells. The register allocator's task is to map these temporary memory cells into a fixed set of n registers, introducing spilling code (load

spill is the one which blocked a fortuitous packing of value lifetimes.

and store) if necessary. Again, it is assumed that data should be kept in registers whenever possible.

Register allocation and spilling via graph coloring consists of the following phases:

(1) live range and U-D (use-definition point) chaining analysis,

(2) interference graph construction, and

(3) interference graph coloring using n colors.

Of these, the first two phases are of little interest with respect to the current work; readers interested in these aspects should refer to [ChA81] [Cha82]. Our primary concern is the actual graph coloring and the register spilling logic involved.

With the interference graph obtained in phase 2, the coloring algorithm reduces the interference graph by taking away all nodes with degree n or less for an n register machine (because they can be colored trivially with n colors). This will continue until, in the ideal case, the graph becomes empty, meaning that the whole graph can be colored with n colors. Nodes are re-attached to the graph in the reverse order and each is assigned a color as it is restored.

If, however, the removal of nodes results in a reduced intermediate graph such that all nodes in the reduced graph have degree greater than n (the number of registers or number of colors available), then the values corresponding to some nodes in the graph must be spilled. This node spilling continues until a graph containing some nodes with degree less than n is obtained. At this point, deletion of nodes of degree less than n continues, as discussed above. This process of deleting nodes and then spilling nodes, thereby introducing stores/loads, continues until the entire graph is reduced to the empty graph (i.e., has been n-colored).

Associated with each node in the graph is a cost estimate computed as the sum of the number of definition points and the number of uses, each weighted by an estimated execution frequency (e.g., a value within a loop has an execution frequency ten times greater than one outside the loop). When a node must be spilled, the node with the lowest ratio of node cost divided by current node degree is chosen.

This model is theoretically more sound than that of the previous approach, since it incorporates data dependence and and a spill cost model. The actual performance of this allocation technique is better than that of

the simple usage count scheme. However, the complexity of implementing this scheme is significantly higher, involving both live range analysis [AhS86] and an NP-complete graph coloring problem — which is why the algorithm uses a heuristic approach to reduce the graph coloring problem complexity to $O(n)$.

Unfortunately, the heuristic used in the coloring algorithm may fail to find an n-coloring for a graph even though one does exist. This introduces unnecessary spills. How often does this occur? The literature is quite vague on this point. Most compilers rarely make use of more than perhaps 5 registers, hence, it is difficult to understand how frequently spills would be forced by faulty colorings of 8, 16, or 32 registers; on the other hand, if register allocation is applied globally, even 32 registers is far too few. For example, given the interference graph as shown in Figure 7.2, the algorithm will fail to two-color the graph, instead adding spill code to the program (since all nodes have degree three, the algorithm would require four colors).

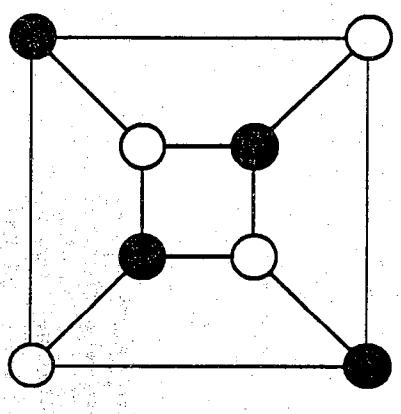


Figure 7.2: Two-colorable Interference Graph

This discrepancy is mainly due to the very crude heuristic of deciding whether the graph can be n-colored. The heuristic used to decide whether a graph can be n-colored is to check if all nodes in the graph have degree greater or equal to n. However, this assumes the worst situation where any node and all its adjacent nodes need to be colored differently (i.e. any node and all its adjacent neighbors are fully connected), which in general is not

true. It is unlikely that all adjacent nodes of a certain node have edges joining each other. In fact, in the best situation adjacent nodes do not interfere with each other and only two colors are necessary.

In addition, register allocation by graph coloring fails in much the same ways that the usage count technique fails. For a machine whose instructions support memory operands, it is important to consider the option of bypassing registers for particular references, yet the graph coloring scheme does not incorporate this alternative.

Further, for any target machine, the cost estimates used are only loosely related to the actual execution-time costs. No ordering of references is taken into account. When there are not enough registers to map all live values, register spilling occurs. Under the register allocation and spilling via coloring scheme, the node with the smallest quotient of estimated cost divided by its current degree is selected. Like the situation in the register allocation via usage count scheme, the value with the smallest current spilling cost may be used immediately. The ratio of the estimated cost of a node divided by its current degree only gives a very rough estimate of the relative cost of spilling values *in the whole basic block* — it does not necessarily give the same decision when a certain segment of program is considered. What is really needed is the information about which value is referenced and when it is referenced, as well as its associated costs (cost and saving) for each step. Optimal register allocation can be achieved only with this kind of information.

7.2.4 Register Allocation by Priority-based Coloring

Register allocation based on some type of priorities is an old concept, dating back to the middle of 1970s [Har75]. This scheme has been suggested and/or implemented by compiler writers to improve the effectiveness of machine registers [Har83].

With the development of RISC architectures, which are register-based machines, the quality of the register allocator can greatly determine the system performance. This has resulted in the development and implementation of the register allocation via priority-based coloring in Stanford MIPS project [ChH84] [Cho83]. Later, this scheme was adapted in the Berkeley SPUR Lisp compiler to improve the register window performance [LaH86].

This scheme is basically a combination of a local register allocation scheme via usage count followed by a global register allocation scheme via graph coloring, along with the introduction of different cost functions to estimate the relative benefits obtained from different values. In the local allocation phase, a register is assigned to one value with the current lowest usage count each time. In the global allocation phase, a register is assigned to one live range with the lowest estimated cost each time. The cost estimates of this algorithm are based on the number of "usedef" in the block and the algorithm tries to estimate the savings when a value is loaded into a register. Loop structures are also considered in this scheme.

To improve the performance of register allocation, this scheme has been extended to permit priorities to be assigned to values in the graph coloring problem, thereby generally improving the spill choices by introducing spill code into code which is executed with a lower frequency. This, however, does not eliminate the basic flaws in the usage count and the standard graph coloring register allocation schemes.

This approach can handle extra bias factors not considered the standard graph coloring approach and the usage count approach. However, since the basic strategies involved are still the same as these two previous algorithms, major problems such as where should a value be referenced and the cost associated with this action still cannot be solved.

7.2.5 Summary of Characteristics of Register Allocations

As a summary, all these register allocation schemes share the following characteristics:

- Register allocation by machine level state transition uses the program's reference sequence in the analysis and this is restricted to a straight line sequence. For other commonly used register allocation schemes, a partial order derived from the reference sequence is used for allocation rather than using the program's actual sequence of references.
- Register allocation is visible to the compiler and, in some cases, also to the programmer.
- In all register allocation schemes, binding of value to a name is defined at compile-time.

- It is well understood that defining live ranges in terms of values rather than in terms of variables is beneficial. The only exception is the register allocation by machine state level transition, where the concept of live range of references is missing.
- Conventional registers are only used to hold data, not instructions.

7.3 Register Allocation By Random Walk Coloring

In the register allocation via graph coloring and spilling [Cha82], the heuristic used for register spilling is: when all nodes in the reduced interference graph have degrees greater than the number of colors (and registers) n available, register spilling occurs until there are some node (s) with degree less than or equal to n.

This heuristic is based on the most conservative argument that a node and all its adjacent neighbors should have different colors. While this argument is safe and the assignment of colors to nodes using this approach is trivial, it is usually far from optimal. In an interference graph, adjacent nodes of some node α may only slightly interfere each other and the number of colors needed for those nodes is usually far less than the degree of α. In the best case, the total number of colors needed for node α and its neighbors might only be two.

We propose a new heuristic, called the **random walk coloring**, to color an interference graph. The idea of this heuristic is actually very simple. Instead of checking the degree of each node and removing nodes from an interference graph, the graph is transversed randomly. Nodes in the interference graph are colored using as few colors as possible. During the graph traversal and coloring process, whenever a visited node cannot be colored (i.e. registers spilling occurs), edges connected to that node α are removed until it can be colored.

As an illustration of the above guidelines, the following subsections present an easily implementable algorithm for random walk coloring. Although good enough to demonstrate its implementation simplicity and to compare with node removal heuristic, this algorithm is far from optimal. There are fine-tuning techniques that can help improving the quality of register assignment. An example is to assign traversal priority to each node based on its expected reference frequency.

Before describing the algorithm, some definitions of notations and terms are helpful:

- Let C be the set of all colors initially available for coloring.
- Let v be the node in the interference graph currently being visited.
- Let $CU(v)$ be the set of colors, each of which has been assigned to at least one *visited* node when node v is currently being visited. $CU(v)$ is always a subset of C and $CU(v)$ is initially is ϕ.
- Let $CA(v)$ be the set of colors, each of which has been assigned to at least one adjacent node of v previously visited when node v is currently being visited. $|CA(v)|$ is always less than or equal to the degree of v and $CA(v)$ is initially ϕ.

7.3.1 Example Algorithm for Random Walk Coloring

In the following discussion of the algorithm for random walk coloring, it is assumed that the interference graph of a program has been found by performing program flow analysis [ChA81] [Cha82].

The algorithm for random walk coloring is as follows:

- A node in a given interference graph is randomly chosen as the starting node for graph traversal.
- The interference graph is traversed randomly. Whenever a node currently being visited has not yet been colored, a color is assinged to it according to the following coloring rules until all nodes in the graph are colored. The coloring rules are:

  $|CU(v)| > |CA(v)|$

  This is the situation where there is at least one color in $CU(v)$ which is used by some nodes in the graph and these nodes are not adjacent to node v. Any color from the set $CU(v)$ - $CA(v)$ can be chosen to color v.

  $|CU(v)| = |CA(v)| < |C|$

  This is the situation where each of the colors currently being used up in the graph has been assigned to at least one adjacent node of node v. However, there are still some colors that have not been used. Any color from the set C - $CU(v)$ can be chosen to color node v.

$|CU(v)| = |CA(v)| = |C|$

This is the situation where all colors are used up by the adjacent nodes of node v and register spilling occurs. One color β is chosen and all edges connecting node v and adjacent node of v with color β assigned to it are cut (i.e. arc spilling). Some possible strategies for choosing the color β for arc spilling are:

- Priority Based

  Nodes in an interference graph are assigned with priorities, which might be based on either the expected reference frequencies of values or the costs of placing values in registers. Guidelines for choosing color β are then based on this reference priority. As an example, the priorities of nodes having the same color are summed up and the color with the minimum total sum is chosen.

- Arc Based

  An alternative approach is to choose color β such that the total number of adjacent nodes of v mapped to color β is minimum.

- Random Based

  In this approach, any color is chosen at random.

In the graph coloring algorithm using random walk, either the depth first search or the breadth first search can be used in the graph traversal. Since each node in the graph is visited once in the graph coloring using random walk, the computational complexity of this algorithm is $O(n^2)$ which is simple enough for practical implementation purposes.

An example might help clarify the algorithm of graph coloring using random walk coloring. Suppose an interference graph is given in Figure 7.3(a) and the total number of colors available is two — color *1* and color *2*. The steps of coloring the graph using random walk coloring are shown in Figure 7.3(b) to Figure 7.3(h). The solid lines in the figures represent the path of graph traversal.

Note that spilling occurs in Figure 7.3(e) and in Figure 7.3(g), where one arc is removed from the graph. If the node removal heuristic is used, a minimum of three nodes have to be spilled.

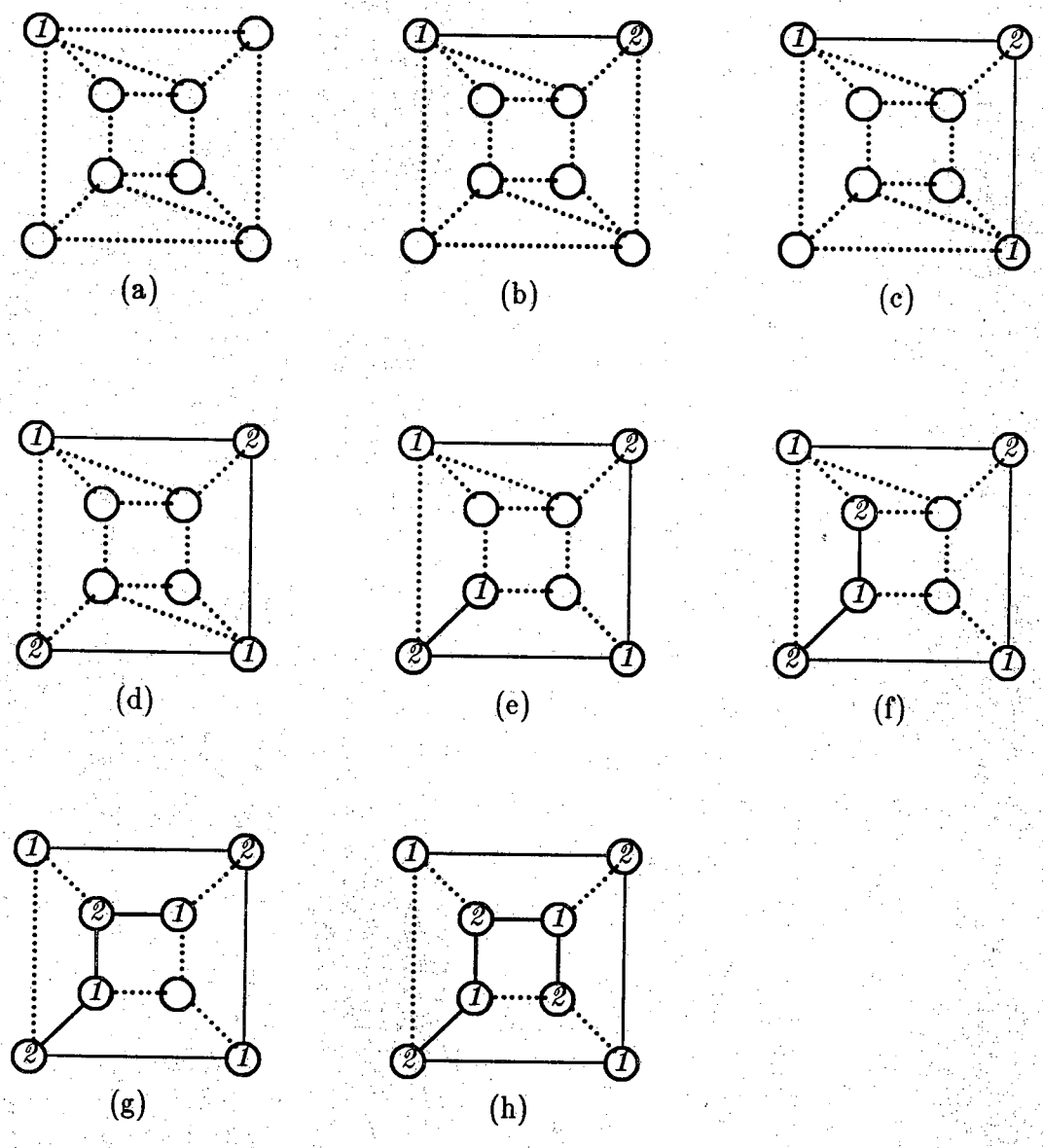


Figure 7.3: Graph Coloring By Random Walk

7.3.2 Comparison of Node Removal and Random Walk Heuristics

Comparing the random walk graph coloring heuristic to the node removal heuristic, we found that:

- The computational complexity of random walk heuristic is $O(n^2)$, where n is the total number of nodes in an interference graph. Hence, random walk heuristic is of the same computational complexity as the node removal heuristic.
- The random walk heuristic is simpler to implement than the node removal heuristic.
- In the node removal heuristic, the relationship between the quality of code generated and the number of registers is monotonically increasing. That is, increasing the number of registers available always decreases the number of register spillings up to the maximum degree of the interference graph. In the random walk heuristic, this relationship only holds most of the time. Due to the random assignment of colors to nodes, it is theoretically possible that simulating the same random walk algorithm with one more register might produce code which is not as good as that produced by the previous simulation with one fewer register. However, this case is very rare and the difference in code qualities is very small.
- A small simulator was built to generate random interference graphs and to color them using these two coloring heuristics. It was found that the random walk heuristic usually performs better than the node removal heuristic and this difference increases as the connectivity of the interference graph increases. As an example, given a graph with about 1000 nodes, with over 60 percent connected, the average number of colors needed by the node removal heuristic without register spilling is from 32 to 40 whereas the random walk heuristic needs only 8 to 12.

7.4 Register Allocation Based on the MAST Model

The MAST model proposed so far describes a compiler-driven management scheme based on machine level state transitions to allocate values used in a program into a high speed buffer — cache. Comparing the features of the high speed buffer used by the MAST model with those of registers (which is summarized in Section 7.2.5), it is clear that they are very

similar. For example, they are all visible to the compiler and benefit from live range analysis.

In this section, the MAST model is extended to perform register allocation. It is found that the same MAST model used in cache management can be applied to register allocation without any change. In fact, additional constraints of registers which cache does not have only simplifies the model.

To see why the MAST model for cache management can be applied to register allocation, it is necessary to understand the MAST views on these two memory structures.

7.4.1 MAST Views on Cache and Registers

The allocation of values to cache using the MAST model has the following characteristics:

- Value allocation is visible to the compiler and is done at compile-time.
- Program reference behavior obtained by static program flow analysis is used in cache management.
- The allocation is based on values of references, not memory addresses.
- The allocation benefits from live range analysis.

Although these characteristics are new to cache management, they have been used by most register allocation schemes. To the MAST model, the registers can be viewed as:

- Registers are used to hold values used in a program, like cache.
- Registers can be viewed functionally as a fully associative cache with line size of one.
- Registers only hold data whereas cache can hold both instruction and data.
- Registers can only hold unambiguous references whereas cache can hold both ambiguous and unambiguous references (This will be discuss in more detail in the Section 7.5).

Hence, registers can be viewed as a special type of cache memory in the MAST model. The fact that only unambiguous data references can be stored in registers only limits the use of the registers. It does not affect the validity of application of the MAST model to register allocation.

7.4.2 MAST Register Allocation

Since the MAST model for register allocation is exactly the same as the one for cache management (and is described in Chapter 3 and 4), it is not repeated here. Instead, an example is used to illustrate how the MAST model might be used to manage either of these two memory structures.

The program segment (see Figure 3.2) used to illustrate the use of the MAST model for cache management is shown in Figure 7.4. Table 7.2 shows the corresponding data reference string. If there are only two registers available, the *feasible register state* after each data reference is shown in Table 7.3. Given the costs for each type of references in Table 7.4, the final MAST graph and the optimal register allocation for this program segment is shown in Figure 7.5. Information about the optimal register control sequence obtained from the analysis is shown in Table 7.5.

$$c = a + b$$
$$d = c * b$$

Figure 7.4: A Program Segment

Table 7.2: The Corresponding Data Reference String

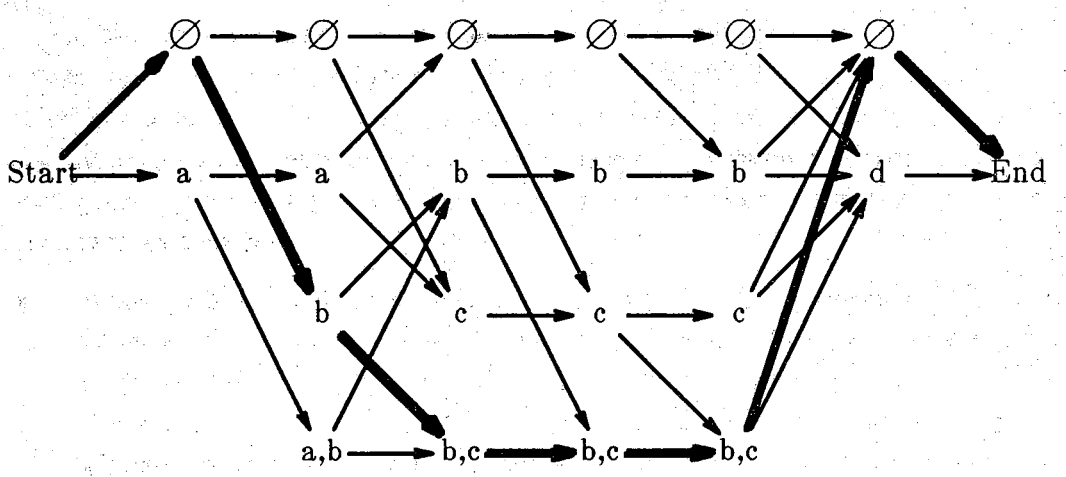
Reference Number	Reference Location of Value	Mode of Reference
1	a	read
2	b	read
3	c	write
4	c	read
5	b	read
6	d	write

Table 7.3: Live Values And Feasible Register States

Value Referenced	Live Values	Feasible Register States
a	a	$\emptyset$,$\{a\}$
b	a,b	$\emptyset$,$\{a\}$,$\{b\}$,$\{a,b\}$
c	b,c	$\emptyset$,$\{b\}$,$\{c\}$,$\{b,c\}$
c	b,c	$\emptyset$,$\{b\}$,$\{c\}$,$\{b,c\}$
b	b,c	$\emptyset$,$\{b\}$,$\{c\}$,$\{b,c\}$
d	d	$\emptyset$,$\{d\}$

Table 7.4. Reference Types and Costs

Reference Type	Name	Read Cost	Write Cost
Register	R	T_{rr}	*N. A.*
Main Memory	M	T_{rm}	T_{wm}
Register, Replacing a Live Non-Dirty or Dead Value	RM	$T_l + T_{rr}$	T_{wr}
Register, Replacing a Live Dirty Value	RMM	$T_s + T_l + T_{rr}$	$T_s + T_{wr}$



Stage 0 Stage 1 Stage 2 Stage 3 Stage 4 Stage 5 Stage 6 Stage 7

Ref. a_r Ref. b_r Ref. c_w Ref. c_r Ref. b_r Ref. d_w

Figure 7.5: Register Allocation by the MAST Model

Table 7.5: Optimal Register Control Sequence

Register State	Data Reference	Register Content after Reference	Reference Mapping	Actions for Reference
Start	—	$\varnothing$	—	—
$v_{1,0}$	a_r	$\varnothing$	mem[a] is a	M-read
$v_{2,2}$	b_r	$\{b\}$	R0 is b	RM-read
$v_{3,3}$	c_w	$\{b,c\}$	R1 is c	RM-write
$v_{4,3}$	c_r	$\{b,c\}$	R1 is c	R-read
$v_{5,3}$	b_r	$\{b,c\}$	R0 is b	R-read
$v_{6,0}$	d_w	$\{d\}$	R0 is d	RM-write

7.5 Unified Registers/Cache Management Model

Register and cache are the top two levels of the traditional computer memory system hierarchy. Although both these structures are used to avoid the delay encountered when processor(s) access main memory, each is traditionally managed by a separate technique. The lack of coordination in managing these two structures results in significant loss of system performance:

- Cache space is wasted to hold inaccessible copies of values in registers.
- Inaccessible copies of values replace those accessible ones from cache.
- Despite the fact that register allocation has long recognized the benefits of live range analysis, traditional cache management has completely ignored live range information. (Note that cache management using the MAST model is the only cache management scheme that includes live range analysis.)

This causes busy redundant memory traffic in cache and decreases system performance substantially. In load/store VLSI processor designs such as RISC architecture [Pat85] [HeJ83] [Kat83], this problem becomes more serious because of the limited on-chip cache size and the high off-chip to on-

chip memory access ratio [Hil83] [AgC87] [KaM87].

In this section, we present an unified scheme for managing registers and cache taking full advantage of live range analysis. Redundant memory traffic in data cache due to inaccessible copies of values are eliminated and cache performance is improved. Throughout the whole discussion of the unified management scheme of registers and cache, a data cache with line size of one is assumed. This assumption is justified by the fact that small line size (e.g. one) is always preferred for data cache [ChD89] [Lee87].

7.5.1 Summary of Differences Between Registers and Cache

In order to devise a coordinated scheme for management of registers and cache and to regain this performance loss, it is first necessary to develop a better understanding of the differences and similarities between these two types of buffer memory.

There is no conceptual difference between the functions of registers and cache in the current memory hierarchy. They might be distinguished by their physical aspects: speed, size, and addressing mode (cache is referenced by address and register by register name). However, from a compiler viewpoint, there are two fundamental conceptual differences between registers and cache:

- Since caches are accessed associatively by main memory addresses, pointer or subscript operations which result in the same memory address being referenced by two different names (aliases) will still reference the same item in cache; this is not true of registers. An aliased value placed in a register will have to be spilled whenever any of its possible aliases is stored into; this spilling makes registers virtually worthless for aliased values [DiC88].
- Since most computers do not have an "execute register" instruction, there is no benefit in placing an instruction in a register.

In summary, registers can be managed more efficiently at compile-time, but cache is far more general in its application. Moreover, the access time of register is faster than that of cache because register is accessed by short register name and cache is accessed by the full memory address. Hence, the ideal is to use registers where they are more efficient, and to use cache *only* for those tasks which cannot benefit from register use. To accomplish this, the traditional cache needs to be modified so that it may at least be

partially controlled by the compiler. For example, a cache bypass bit (as is discussed in Chapter 5) is included.

7.5.2 A Unified View of Cache and Registers

In the previous section we have characterized the differences between registers and cache as primarily differences in the types of items which can be profitably kept in each. In this section, the complete strategy for managing registers and cache using a coordinated scheme is proposed. The key idea of this scheme is to try to keep only one copy of information in either cache or registers. Hence, any inaccessible copy of information can be eliminated and the effectiveness of each memory level increases.

Perhaps the best summary is the diagram of Figure 7.6. As depicted in Figure 7.6, the unified registers/cache management model is fundamentally different from previous proposals in that it takes full advantage of the conceptual differences between registers and cache.

From the compiler's view, memory references in a program can be classified into three different types:

- ambiguous data values,
- unambiguous data values, and
- instructions.

To avoid any inaccessible copies of values in the local memory hardware, any placement of memory reference values should be done according to the **usability**[2] of each memory level.

Registers are very restricted in their usability, and register allocation techniques such as graph coloring [ChA81] [Cha82] [Cho83] or the MAST model can be used in the domain of unambiguous value references. Here, we propose that the conventional management techniques be used, but with three differences:

- When a register will be used for a series of operations, the loading and storing of the value into a register should bypass the cache.

---

[2] The usability of a memory level is defined as the capability of a memory level to handle some particular type of information very *efficiently*.

- When a register's value must be spilled due to a shortage of registers, it should be spilled to cache.
- When the spilled value is referenced, it is either reloaded from:
  - Cache

    In this case, the cached copy becomes dead as soon as the value is reloaded into a register.
  - Main Memory

    In this case, the cache is bypassed and the value is directly referenced from the main memory.

The subdivision of references into ambiguous values, unambiguous values, and instructions is relatively straightforward compiler technology; a brief description appears in [DiC88]. Hence, determining which references should be handled by register allocation and which by cache management is a simple matter. Since register spills should go to cache, however, there is a natural ordering that register allocation should precede cache management decisions.

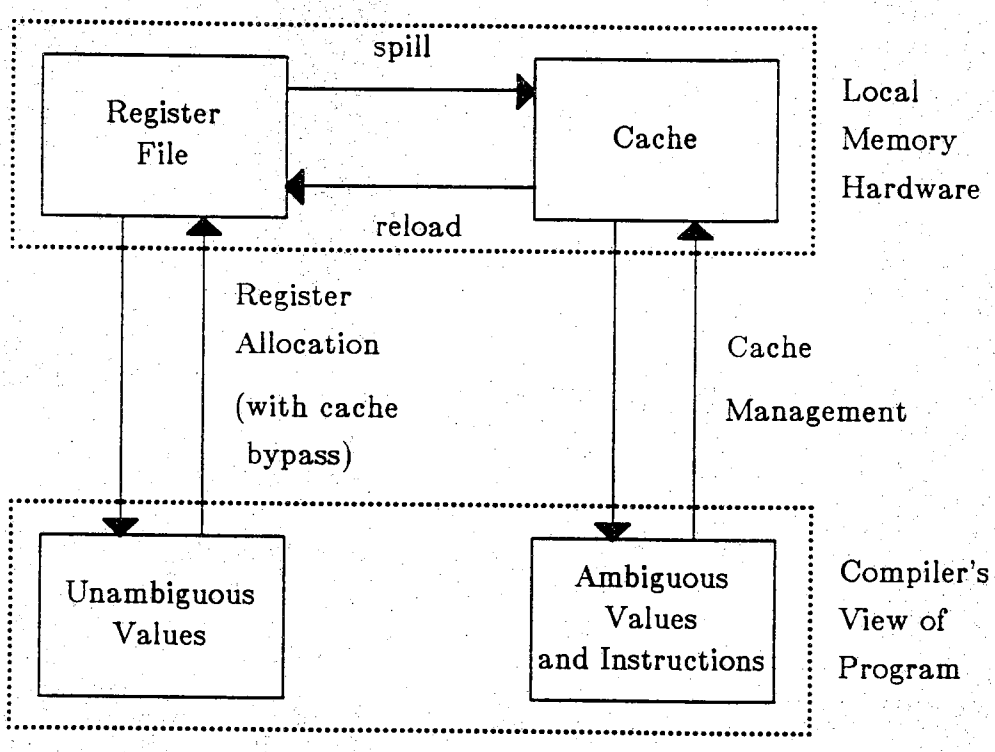


Figure 7.6: Unified Registers/Cache Management Model

7.5.3 Semantics for the Unified Model

The following semantics are defined for a register-to-register operation architecture. However, they can easily be extended to other types of architectures with slight modifications.

With the unified registers/cache management model, there are four different types of load/store instructions corresponding to the fetching and storing of values in cache and registers. They are:

- Am_LOAD,
- AmSp_STORE,
- UmAm_LOAD, and
- UmAm_STORE.

A cache bypass bit per each memory reference is also used to indicate if the reference goes through the cache. A "1" would means "bypass" and a "0" means "go through the cache".

The operations of these four load/store instructions are as follows:

- **Am_LOAD**

  This type of load instruction fetches a datum into a register through cache. That is, a copy of the datum will appear in cache after the reference and the cache bypass bit is set to zero. This instruction is used for loading ambiguous values.

- **AmSp_STORE**

  This type of store instructions saves a datum through cache. That is, the datum is placed in cache and the cache bypass bit is set to zero. There are two situations which use this store instruction:

  - when an ambiguous value is stored.
  - when an unambiguous value is spilled from a register.

- **UmAm_LOAD**

  The operation of this instruction is to check if the datum is cache. If it is in cache, the datum is loaded into a register and that datum in cache is then marked as invalid or empty. If it is not in cache, the datum is loaded from main memory to registers directly, bypassing the cache. In both cases, the cache bypass bit is set to one. This type of load instruction is used for loading unambiguous values.

- **UmAm_STORE**

  This type of store instruction saves a datum directly to main memory,

bypassing the cache. The cache bypass bit is set to one. It is used for saving unambiguous values into main memory which are not due to register spilling.

7.6 Conclusion

In this chapter, the MAST model proposed in previous chapter is extended to handle register allocation problem. It is found that the same model can be applied to both registers and cache and the version of the MAST model for register allocation is even simpler than that for cache management.

Registers and cache are *not* interchangeable, but are complementary to each other. A machine with 1000000 registers would not be able to place all values in registers, because registers cannot resolve ambiguously aliased references. A machine with 1000000 words of cache but no registers could, however, be equally futile in that, without the compile-time management associated with registers there is no provision for avoiding worst-case cache scenarios. In some cases, the machine would spend more time placing lines in cache and referencing them there than it would spend performing references directly from main memory (faster without cache than with it); even discounting that effect, cache access time is nearly always longer than register access time so using cache where registers would suffice is not optimal.

Miller found that the ratio of unambiguous references to ambiguous references, measured statically, is from 1:1 to 3:1 [Mil88]. This does suggest that registers are more important than cache, however, it does not count instruction references. Hence, the load placed on each type of memory is considerable.

Given these surprising realizations, we have proposed a coordinated registers/cache management scheme which can use each hardware structure for the references for which it is best suited. This technique is both implementable and familiar, being very closely related to register allocation.

CHAPTER VIII
CONCLUSIONS

This thesis presents an alternative methodology for the modeling management of memory, especially cache and register. This model, based on a graph formalism using state transitions and arbitrary cost functions, not only promises that optimal transaction sequences can be found, but also that the compiler can explicitly manage runtime activities to achieve that sequence. It also provides a previously unavailable way of improving cache performance — a way in which large performance increases are likely to be achieved.

We conclude this thesis by summarizing its main contributions to memory hierarchy design and management. The significance of each contribution is discussed and its possible applications are given whenever possible. Since this research provides a *new and accurate* approach understand and manage cache and registers, many ideas for possible future research directions are inspired. We present some of these ideas as the conclusion of this chapter.

8.1 Primary Research Contributions

The primary research contributions of this thesis can be summarized as follows:

- Cache Bypass

  In traditional cache designs, one of the most fundamental concepts is the "cache through" — it is *always* beneficial to place as much information in cache as possible. This constraint results in a significant loss of system performance because infrequently referenced information is forced into cache. In multiprocessor systems, where the reference delay for global information is very large, this system performance loss becomes more serious. Being able to selectively bypass the cache can improve the cache in a way that none of the current cache management

schemes can provide. Further, the hardware and software needed for this selective cache bypass is simple enough to be used immediately.

- Live Range of References in Cache

  Although register allocation schemes have benefited from live range analysis of values for a long time, the concept of live range of references in cache is completely missing in current cache designs. With the correct definition of live range of references in cache, program reference behavior can be understood and explained more clearly and accurately. This also helps to improve cache performance by removing "dead" information from cache.

- Cost Consideration

  Since the cost for each cache state transition is considered, computers with distributed and hierarchical memory systems can be correctly modeled and performance drastically improved. In a multiprocessing environment, this is especially important because the difference between storing and/or referencing from local memory relative to global memory makes a big difference in the final cache performance. This is a big step forward since no previous technique employed *any* model of these costs.

- Measuring Parameter

  The cache hit ratio has always been considered as the most important performance parameter to be optimized in cache designs. It is assumed that cache hit ratio is directly related to system performance. However, results from our research show that this is not always true. When other execution time related parameters such as cache line size are varied, system performance changes. Hence, the cache hit ratio should be used only if all other execution time related parameters remain unchanged. Furthermore, this research suggests that total memory reference time of a program should be used as the performance parameter to be measured and optimized. This is because the total memory reference time combines the effects of all execution time related cache parameters and is a direct reflection of system performance.

- Cache Line Size Selection

  Using total reference time as the measuring parameter for cache designs, it was found that a small cache line size (e.g. one or two) is usually preferred. The explanation for this cache line size selection is based on the cost of cache operations. This provides a better understanding of the effect of cache line size on cache performance and gives a very useful guideline for cache designs. None of the current cache management schemes can provide this information.

- Upper Bound Cache Performance

  Given a cache design, the MAST Model can provide an upper bound on the performance of the cache on a given program or application. This is extremely important because with this performance standard, the effectiveness of any other cache policies and their potential improvements can be measured and compared.

- Cache Performance Improvement

  Using the MAST Model for controlling cache activities, optimal or nearly optimal cache performance can be obtained without increasing cache hardware complexity. The MAST Model provides a very practical, implementable, and effective method to control cache activities by the compiler. Based on the state transition analysis, the MAST Model can improve cache performance in the ways that none of the traditional cache policies (e.g. LRU, Random, FIFO) can.

- Heuristics for Register Allocation

  Heuristics for register allocation using graph coloring were studied and re-evaluated. A new heuristic, called the random walk, was also presented as a simpler and more effective way to color an interference graph. Simulation results show that random walk heuristic produces better code quality than node removal.

- Unified Cache/Registers Management

  Given a local memory hierarchy of registers and cache, the unified cache/registers model improves the cache performance by removing redundant copies of values in cache. The bus traffic and the memory traffic in data cache are greatly reduced and the cache effectiveness is

increased. None of the current cache and/or register management schemes can improve cache performance or bus traffic in this way.

- Model for Other Levels of Memory Hierarchy

  Finally, this MAST Model based on state transitions is powerful enough to be applied to other levels of memory hierarchies. For example, an optimal register allocation scheme has been derived from this MAST Model and it is shown that it has better performance than any other previous register allocation schemes. Similar derivations might also be applied to virtual memory or multilevel cache designs.

8.2 Future Research Directions

There are a lot of research areas and extensions that are suggested by this research efforts. Some of the more immediate ones are as follows:

- Cache Prefetching

  Cache prefetching has been considered as one of the effective way to improve cache performance. However, the main problem for cache prefetching is cache pollution. With program flow analysis as a necessary step for any version of the compiler-driven cache management schemes, future references can be predicted more accurately. This information is extremely valuable to cache prefetching schemes because cache pollution problem is almost solved and instruction/data can be prefetched into the cache only if they are known to be needed. Hence, compiler-driven cache prefetching should be considered as the next major extension of this research.

- Compiler Optimization to Cache Performance

  Throughout this research, the main effort has been to obtain the best performance from a given cache configuration and a *fixed* reference program. Since the compiler plays an important role in the management scheme, a natural way to further improve cache performance is to perform program transformation. There are at least three different ways to do this:

  - Locality of references can be improved by grouping information that are referenced simultaneously in the same cache line.

- Memory conflict can be reduced by distributing references that are made simultaneously across different memory modules.
- The mapping of memory addresses to values for instructions and/or data can be re-defined so that the total number of lines needed to be fetched for program execution is reduced.

• Impact to Architecture Design

With this compiler-driven cache management scheme, a small cache can be placed onto the same processor chip (even in the case of GaAs 32-bit RISC processor [DiC88] where the chip area is very expensive) to bridge the reference delay gap efficiently. This will have a big impact on the architecture design because the problem of on-chip to off-chip memory reference bottleneck changes. Hence, some of the architecture design concepts need to be re-evaluated and modified if necessary. Some good examples are the impact of the on-chip cache to pipeline design and delay fill-in by no-ops using explicit cache control instructions.

• Virtual Memory Design

The MAST model proposed here is mainly designed for register allocation and cache management schemes. However, most of the ideas presented here can also be applied to virtual memory to improve the system performance. Furthermore, since the same model can be applied to all levels of the memory hierarchy, an unified management scheme for the whole memory hierarchy — registers, cache, and virtual memory — is possible to further improve the system performance.

LIST OF REFERENCES

LIST OF REFERENCES

[Abu79] Abu-Sufah, W.A., "Improving the Performance of Virtual Memory Computers," *Ph.D Thesis*, University of Illinois at Urbana-Champaign, 1979.

[AgC87] Agarwal, A., Chow, P., Horowitz, M., Acken, J., Salz, A., Hennessy, J., "On-Chip Instruction Caches for High Performance Processors," *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI*, edited by Losleben, P., The MIT Press, 1987, pp. 1-24.

[Agr79] Agresti, W.W., "Register Assignment in Tree-Structured Program," *Information Sciences*, Volume 18, Number 2, July, 1979, pp. 83-94.

[AhH74] Aho, A.V., Hopcroft, J.E., Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, Massachusetts, 1974.

[AhS86] Aho, A.V., Sethi, R., Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, Massachusetts, 1986.

[AhU77] Aho, A.V., Ullman, J.D., *Principles of Compiler Design*, Addison Wesley, Reading, Massachusetts, 1977.

[AlB86] Allen, R., Baumgartner, D., Kennedy, K., Porterfield, A., "PTOOL: A Semi-Automatic Parallel Programming Assistant," *1986 International Conference on Parallel Processing*, August 1986, pp. 164-170.

[AlC76] Allen, F.E., Cocke, J., "A Program Data Flow Analysis Procedure," *Communication of the ACM*, Volume 19, 1976, pp. 137-147.

[AlW75] Alexander, G., Wortman, D., "Static and Dynamic Characteristics of XPL Programs," *IEEE Computer*, Volume 8, Number 11, November 1975, pp. 505-525.

[Bab82] Babaoglu, O., "Hierarchical Replacement Decisions in Hierarchical Stores," *Proceeding of the 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1982, pp. 11-19.

[Bac57] Backus, J. W., Et. Al., "The Fortran Automatic Coding System," *Proceedings of the Western Joint Computer Conference*, Volume 11, 1957, pp. 188-198.

[Bal79] Ball, J.E., "Predicting the Effects of Optimization on a Procedure Body," *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, SIGPLAN Notice*, 1979.

[BaS76] Baer, J.L., Sager, G.R., "Dynamic Improvement of locality in virtual memory systems," *IEEE Transactions on Software Engineering*, Volume SE-2, Number 1, 1976, pp. 54-62.

[Bel66] Belady, L.A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM System Journal*, Volume 5, 1966, pp. 78-101.

[Bel74] Belady, L.A., Palermo, F.P., "On-line Measurement of Paging Behavior by the Multi-valued MIN Algorithm," *IBM Research and Development*, Volume 18, Number 1, January, 1974, pp. 2-19.

[Bre87] Brent, G.A., "Using Program Structure to Achieve Prefetching for Cache Memories," *Ph.D Thesis*, University of Illinois at Urbana-Champaign, January 1987.

[BuC86] Burke, M., Cytron, R., "Interprocedural Dependence Analysis and Parallelization," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notice* 1986, pp. 162-175.

[ChA81] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W., "Register Allocation Via Coloring," *Computer Languages*, Volume 6, 1981, pp. 47-57.

[Cha82] Chaitin, G.J., "Register Allocation and Spilling via Graph Coloring," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notice* Volume 17, Number 6, June 1982, pp. 201-207.

[ChD87] C.H. Chi, Dietz, H., "Compiler-Driven Cache Policy," *Purdue University Technical Report TR-EE 87-21*, 1987.

[ChD88] C.H. Chi, Dietz, H., "Register Allocation for GaAs Computer Systems," *Proceedings of the 1988 Hawaii International Conference on Systems Sciences*, January 1988, pp. 266-274.

[ChD89] Chi, C.H., Dietz, H., "Improving Cache Performance by Selective Cache Bypass," *Proceeding of the 1989 Hawaii International Conference on Systems Sciences,* January 1989, pp. 256-265.

[ChH84] Chow, F., Hennessy, J., "Register Allocation by Priority-based Coloring," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notice* Volume 19, Number 6, June 1984, pp. 222-232.

[Cho83] Chow, F.C., "A Portable Machine-Independent Global Optimizer-Design and Measurement," *Technical Note no. 83-254*, Computer Systems Laboratory, Stanford University, December 1983.

[ChS86] Cheriton, D.R., Slavenburg, G.A., Boyle, P.D., "Software-Controlled Caches in the VMP Multiprocessor," *Report no. STAN-CS-86-1105*, Department of Computer Science, Stanford University, March 1986.

[Con86] "C1 Processor Series: Architecture," Convex Computer Corporation, 1986.

[Coo84] Cooper, K.D., "Analyzing Aliases of Reference Formal Parameters," *Proceeding of the* 11th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1984, pp. 281-290.

[Cou86] Coutant, D.S., "Retargetable High-Level Alias Analysis," *Proceeding of the* 13th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1986, pp. 110-118.

[Den68] Denning, P.J., "The Working Set Model for Program Behavior," *Communications of the ACM*, Volume 11, Number 5, May 1968, pp. 323-333.

[Den70] Denning, J.P., "Virtual Memory," *ACM Computing Surveys*, Volume 2, Number 3, September 1970, pp. 62-97.

[DiC88a] Dietz, H., Chi, C.H., "A Compiler-Writer's View of GaAs Computer System Design," *Proceedings of the 1988 Hawaii International Conference on Systems Sciences*, January 1988, pp. 256-265.

[DiC88b] Dietz, H., Chi, C.H., "CRegs: A New Kind of Memory for Referencing Arrays and Pointers," *Proceeding of the Supercomputing'88 Conference*, November, 1988.

[Die87] Dietz, H.G., "The Refined-Language Approach to Compiling for Parallel Supercomputers," *Ph.D. Thesis*, Polytechnic University, June 1987.

[DoJ79] Dongarra, J.J., Jinds, A.R., "Unrolling Loops in Fortran," *Software Practice and Experience*, Volume 9, Number 3, March 1979, pp. 219-226.

[DuB82] Dubois, M., Briggs, F.A., "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers*, Volume C-31, Number 11, November, 1982, pp. 1083-1099.

[Ell85] Ellis, J.R., *Bulldog: A Compiler for VLIW Architectures, Ph.D Thesis*, Yale University, MIT Press, 1985.

[Fre74] Freiburghouse, R.A., "Register Allocation Via Usage Courts," *Communications of the ACM*, Volume 17, Number 11, November 1974, pp. 638-642.

[Fis81] Fisher, J.A., "Trace Scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, Volume C-30, Number 7, July 1981, pp. 478-490.

[GoH86] Goodman, J.R., Hsu, W.C., "On the Use of Registers Vs. Cache to Minimize Memory Traffic," *Proceeding of the* 13th Annual International Symposium on Computer Architecture, June, 1986, pp. 375-383.

[Har75] Harrison, W., "A Class of Register Allocation Algorithms," *Computer Center Technical Report RC 5342*, IBM Thomas J. Watson Research Center, March 27, 1975.

[Har83] Hartmann, A., "iAPX286 Compiler Writer's Guide," *Version #1, Intel Corporation*, May 1983.

[HeJ83] Hennessy, J., Jouppi, N., Baskett, F., Gill, J., "MIPS: A VLSI Processor Architecture," *Technical Report No. 223, Computer Systems Laboratory*, Stanford University, June 1983.

[Hil83] Hill, M.D., "Evaluation of On-Chip Cache," *M.S. Thesis*, University of California, Berkeley, December, 1983.

[HiS84] Hill, M.D., Smith, A.J., "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proceeding of the* 11th Annual International Symposium on Computer Architecture, June 1984, pp. 158-166.

[HoK66] Horwitz, L.P., Karp, R.M., Miller, R.E., Winograd, S., "Index Register Allocation," *Journal of the ACM*, Volume 13, Number 1, January 1966, pp. 43-61.

[Hol88] Holman, R., *personal communication*, June 1988.

[Hsu85] Hsu, W.C., "Register Allocation for VLSI Processors," *UW Computer Science Technical Report #619*, November, 1985.

[Hsu87] Hsu, W.C., "Register Allocation and Code Scheduling for LOAD/STORE Architectures," *Ph.D Thesis*, Computer Science Department, University of Wisconsin, Madison, 1987.

[HwB84] Hwang, K., Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw Hill Book Company, 1984.

[Int86] Intel Corporation, *80386 programmer's reference manual*, 1986, pp. 11-6.

[Joh77] Johnson, D.B., "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, Volume 24, Number 1, January 1977, pp. 1-13.

[Jos70] Joseph, M., "An Analysis of paging and program behavior," *Computer Journal*, Volume 13, Number 1, 1970, pp. 48-54.

[Kab87] Kabakibo, A., "Cache Design in Silicon and GaAs Microcomputer Systems," *M.S. Thesis*, School of Electrical Engineering, Purdue University, May 1987.

[KaM87] Kadota, H., Miyake, J., Okabayashi, I., Maeda, T., Okamota, T., Nakajima, M., Kagawa, K., "A 32-bit CMOS Microprocessor with On-Chip Cache and TLB," *IEEE Journal of Solid-State Circuits*, Volume SC-32, Number 5, October 1987, pp. 800-807.

[Kat83] Katevenis, M.H., "Reduced Instruction Set Computer Architectures for VLSI," *Ph.D Thesis*, Department of Electrical and Computer Science, University of California, Berkeley, 1983.

[Ken71a] Kennedy, K., "Global Flow Analysis and Register Allocation for Simple Code Structures," *Ph.D Thesis*, Department of Computer Science, New York University, 1971.

[Ken71b] Kennedy, K., "Index Register Allocation in Straight Line Code and Simple Loops," *Design and Optimization of Compilers*, edited by Rustin, R., Prentice-Hall Inc., 1971, pp. 51-63.

[Kob80] Kobayashi, M., "Instruction Reference Behavior and Locality of Reference in Paging," *Ph.D Thesis*, University of California, Berkeley, June 1980.

[Kob83] Kobayashi, M., "Dynamic Profile of Instruction Sequences for the IBM System/370," *IEEE Transactions on Computers*, Volume C-32, Number 9, September 1983, pp. 859-861.

[Kob84] Kobayashi, M., "Dynamic Characteristics of Loops," *IEEE Transactions on Computers*, Volume C-33, Number 2, February 1984, pp. 125-132.

[LaH86] Larus, J.R., Hilfinger, P.N., "Register Allocation in the SPUR Lisp Compiler," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notice* Volume 21, Number 6, June 1986, pp. 255-263.

[Lee87] Lee, R.L., "The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors," *Ph.D Thesis*, University of Illinois at Urbana-Champaign, May 1987.

[LeS84] Lee, K.F., Smith, A.J., "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, Volume 17, Number 1, January, 1984, pp. 6-22.

[LeY87] Lee, R.L., Yew, P.C., Lawrie, D.H., "Multiprocessor Cache Design Considerations," *Proceeding of the* 14th Annual International Symposium on Computer Architecture, June 1987, pp. 253-263.

[Lil88] Lilja, D.J., "Reducing the Branch Penalty in Pipelined Processors," *IEEE Computer*, Volume 21, Number 7, July, 1988, pp. 47-55.

[Lip68] Liptay, J., "Structural Aspects of the System/360 Model 85: Part II: The Cache," *IBM Systems Journal*, 1968, pp. 15-21.

[Luc67] Luccio, F., "A Comment on Index Register Allocation," *Communications of the ACM*, Volume 10, Number 9, September, 1967, pp. 572-574.

[Lun77] Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Communications of the ACM*, Volume 20, Number 3, March 1977, pp. 143-153.

[Mac83] Mace, M.E., "Globally Optimum Selection of Memory Storage Patterns," *Ph.D Thesis*, Duke University, 1983.

[Mac87] Mace, M.E., *Memory Storage Patterns in Parallel Processing*, Kluwer Academic Publishers, 1987.

[McD88] McNiven, G.D., Davidson, E.S., "Analysis of Memory Referencing Behavior For Design of Local Memories," *Proceeding of the* 15th Annual International Symposium on Computer Architecture, June, 1988, pp. 56-63.

[MiF86] Milutinovic, V., Fura, D., Helbig, W., "An Introduction to GaAs Microprocessor Architecture for VLSI," *IEEE Computer*,

Volume 19, Number 3, March 1986, pp. 30-42.

[MiF87] Milutinovic, V., Fura, D., Helbig, W., Linn, J., "Architecture/Compiler Synergism in GaAs Computer Systems," *IEEE Computer*, Volume 20, Number 5, May 1987, pp. 72-93.

[Mil88] Miller, B.P., "The Frequency of Dynamic Pointer References in "C" Programs," *Computer Sciences Technical Report #759*, University of Wisconsin, Madison, March 1988.

[Mit86a] Mitchell, C.L., "Processor Architecture and Cache Performance," *Ph.D Thesis*, Stanford University, June 1986.

[Mit86b] Mitchell, C.L., "Architecture and Simulation Results for Individual Benchmarks," *Technical Note: CSL-TN-86-289*, Stanford University, December 1986.

[NeP87] Neirynck, A., Panangaden, P., "Computation of Aliases and Support Sets," *Proceeding of the* 14th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1987, pp. 274-283.

[Nem66] Nemhauser, G.L., *Introduction to Dynamic Programming*, Wiley, New York, 1966.

[PaG83] Patterson, D.A.,, Garrison, P., Hill, M., Lioupis, D., Nyberg, C., Sippel, T., Dyke, K.V., "Architecture of a VLSI Instruction Cache for a RISC," *Proceeding of the* 10th Annual International Symposium on Computer Architecture, June, 1983, pp. 108-116.

[PaS82] Patterson, D., Sequin, C., "A VLSI RISC," *IEEE Computer*, Volume 15, Number 9, September, 1982, pp. 8-21.

[Pat85] Patterson, D.A., "Reduced Instruction Set Computers," *Communications of the ACM*, Volume 28, Number 1, January 1985, pp. 8-21.

[PeS85] Peterson, J. L., Silberschatz, A., *Operating Systems Concepts*, Addison-Wesley Publishing Company, 1985, pp. 222-226.

[PoA83] Pohm, A.V., Agrawal, O.P., *High-Speed Memory Systems*, Reston Publishing Company, Inc., 1983.

[PrH88] Przybylski, S., Horowitz, M., Hennessy, J., "Performance Tradeoffs in Cache Design," *Proceeding of the* 15th Annual International Symposium on Computer Architecture, June, 1988, pp. 290-298.

[Puz85] Puzak, T.R., "Analysis of Cache Replacement Algorithms," *Ph.D Thesis*, University of Massachusetts, 1985.

[Rad83] Radin, G., "The 801 Minicomputer," *IBM Journal of Research and Development*, May 1983, pp. 237-246.

[Sch73] Schwartz, J.T., *On Programming: An Interim Report on the SETL Project*, Courant Institute of Math. Sciences, New York University, 1973.

[Sch77] Scheifler, R.W., "An Analysis of Inline Substitution for a Structured Programming Language," *Communication of the ACM*, Volume 20, Number 9, September 1977, pp. 647-654.

[Sit79] Sites, R., "How to Use 1000 Registers," *Proceeding of the Caltech Conference on VLSI*, January 1979, pp. 527-536.

[SmG83] Smith, J.E., Goodman, J.R., "A Study of Instruction Cache Organizations and Replacement Policies," *Proceeding of the* 10th Annual International Symposium on Computer Architecture, June 1983, pp. 132-137.

[SmG85] Smith, J.E., Goodman, J.R., "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers*, Volume C-34, Number 3, March 1985, pp. 234-241.

[Smi78a] Smith, A.J., "Sequentiality and Prefetching in data base systems," *ACM Transactions on Data Base Systems*, Volume 3, Number 3, 1978, pp. 223-247.

[Smi78b] Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Volume 11, Number 12, 1978, pp. 7-21.

[Smi78c] Smith, A.J., "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Transactions on Software Engineering*, Volume 4, Number 2, March 1978, pp. 121-130.

[Smi82] Smith, A.J., "Cache Memories," *Computing Surveys*, Volume 14, Number 3, September, 1982, pp. 473-530.

[Smi87a] Smith, A.J., "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, Volume C-36, Number 9, September 1987, pp. 1063-1075.

[Smi87b] Smith, A.J., "Cache Memory Design: An Evolving Art," *IEEE Spectrum*, Volume 24, Number 12, December 1987, pp. 40-44.

[SoR88] So, K.M., Rechtschaffen, R.N., "Cache Operations by MRU Change," *IEEE Transactions on Computers*, Volume 37, Number 6, June 1988.

[Spa74] Spaniol, 0., "Demand Prepaging Algorithms Basing on a Model of Locality of Programs," *Computer Architectures and Networks*, edited by Gelenbe, E., Mahl, R., North-Holland, Amsterdam, 1974, pp. 515-527.

[Spi76] Spirn, J., "Distance String Models for Program Behavior," *IEEE Computer*, November, 1976, pp. 14-20.

[Spi77] Spirn, J., *Program Behavior: Models and Measurements*, Elsevier-North Holland, N.Y., 1977.

[Ste86] Stein, K., *Refined C Compiler Status Report*, Internal Report, Stevens Institute of Technology, 1986.

[Sto87] Stone, H.S, *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, 1987, pp. 21-101.

[StS85] Steele, G.L. Jr., Sussman, G.J., "The Dream of a Lifetime: A Lazy Variable Extent Mechanism," *ACM SIGPLAN order number 552800*, 1985, pp. 163-172.

[Tan78] Tanenbaum, A., "Implications of Structured Programming for Machine Architecture," *Communications of the ACM*, Volume 21, Number 3, March 1978, pp. 237-246.

[Tha81] Thabit, K.D., "Cache Management by the Compiler," *Ph.D Thesis*, Rice University, November, 1981.

[Vei85] Veidenbaum, A.V., "Compiler Optimizations and Architecture Design Issues for Multiprocessors," *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, May 1985.

[Vei86] Veidenbaum, A.V., "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *IEEE no. 0190-3918/86/0000/1029*, 1986, pp. 1029-1036.

[Wag76] Wagner, R.A., "A Shortest Path Algorithm for Edge-Sparse Graphs," *Journal of the ACM*, Volume 23, Number 1, January 1976, pp. 50-57.

[Yer71] Yershov, A.P., *The Alpha Automatic Programming System*, Academic Press, London, 1971.

VITA

Chi-Hung Chi was born in April 7, 1961 in Hong Kong. In August 1982, he went to study in the University of Wisconsin - Madison, majoring in computer engineering. In May 1984, he received his Bachelor of Science in Electrical Engineering. He continued his graduate study at Purdue University, majoring in Computer Architecture and Compiler Optimization. On September 5, 1987 he was married to Lai-Fan Yu. His major research areas include interaction between compiler and architecture, design and modelling of various levels of memory hierarchies (particularly cache and registers), RISC architecture designs and tradeoffs, optimizing compiler, and parallel processing. He is currently working at North American Philips Corporations. He can be reached by:

Dr. Chi-Hung Chi
North American Philips Corporations
345 Scarborough Road
Briarcliff Manor, NY 10510
Phone: (914) 945-6498
E-mail: chc@philabs.philips.com